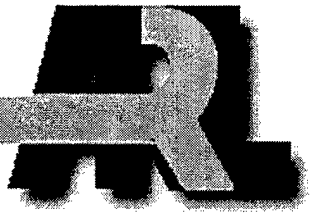
ARMY RESEARCH LABORATORY



Common High Performance Computing Software Support Initiative (CHSSI) Computational Fluid Dynamics (CFD)-6 Project Final Report: ARL Block-Structured Gridding Zonal Navier-Stokes Flow (ZNSFLOW) Solver Software

Harris L. Edge
Jubaraj Sahu
Walter B. Sturek
Daniel M. Pressel
Karen R. Heavey
Paul Weinacht
Csaba K. Zoltani
Charles J. Nietubicz
Jerry Clarke
Marek Behr
Patrick Collins

ARL-TR-2084 FEBRUARY 2000

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5066

ARL-TR-2084 February 2000

Common High Performance Computing Software Support Initiative (CHSSI) Computational Fluid Dynamics (CFD)-6 Project Final Report: ARL Block-Structured Gridding Zonal Navier-Stokes Flow (ZNSFLOW) Solver Software

Harris L. Edge
Jubaraj Sahu
Karen R. Heavey
Paul Weinacht
Weapons & Materials Research Directorate, ARL

Walter B. Sturek
Daniel M. Pressel
Csaba K. Zoltani
Charles J. Nietubicz
Corporate Information & Computing Directorate, ARL

Jerry Clarke
Raytheon Systems Company

Marek Behr
Rice University

Patrick Collins
U.S. Department of the Treasury

Abstract

This report presents an overview of the software developed under the common high performance computing software support initiative (CHSSI), computational fluid dynamics (CFD)-6 project. Under the project, a zonal Navier-Stokes flow solver tested and validated via years of productive research at the U.S. Army Research Laboratory was rewritten for scalable parallel performance on both shared memory and distributed memory high performance computers. At the same time, a graphical user interface (GUI) was developed to help the user set up the problem, provide real-time visualization, and execute the solver. The GUI is not just an input interface but provides an environment for the systematic, coherent execution of the solver, thus making it a more useful, quicker and easier application tool for engineers. Also part of the CHSSI project is a demonstration of the developed software on complex applications of interest to the Department of Defense (DoD). Results from computations of 10 brilliant antitank (BAT) submunitions simultaneously ejecting from a single Army tactical missile and a guided multiple launch rocket system missile are discussed. Experimental data were available for comparison with the BAT computations. The CFD computations and the experimental data show good agreement and serve as validation for the accuracy of the solver. The software has been written with large memory requirements and scalability in mind. For a grid size of 59 million points, the performance achieved on an Silicon Graphics, Incorporated, Origin 2000 with 96 processors is 18 times the performance that could be achieved via a computer with the processing speed of a single Cray C-90 processor.

ACKNOWLEDGMENTS

The authors would like to thank the High Performance Computing Center at the U.S. Army Research Laboratory's major shared research center at Aberdeen Proving Ground, Maryland, and the Naval Research Laboratory distributed center in Washington, DC, for the use of their computing resources.

An acknowledgment is given to others who at some point participated in the zonal Navier-Stokes flow (ZNSFLOW) user meetings and provided feedback and/or input which helped in the development of the ZNSFLOW software. Those who have participated are Clint Housh of the Naval Air Warfare Center, Mark McKelvin, and Olu Olatidoye of Clarke Atlanta University in Georgia, Steve Scherr of the Air Force Research Laboratory in Florida, and Andrew Wardlaw of the Naval Surface Weapons Center in Virginia.

The authors appreciate the support of the Department of Defense common high performance computing software support initiative (CHSSI) program members and evaluators. An acknowledgment is given to those participants in the CHSSI program directly involved with the ZNSFLOW project: Jay Boris, the computational fluid dynamics critical technical area leader for CHSSI and ZNSFLOW alpha tester, and David Fife, ZNSFLOW alpha and beta tester.

The authors would like to express their appreciation for the time and expertise of Dixie Hisley and Nancy Nicholas for their technical and editorial reviews, respectively.

INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

INTENTIONALLY LEFT BLANK

LIST OF FIGURES

INTENTIONALLY LEFT BLANK

LIST OF TABLES

INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Under the auspices of the common high performance computing software support initiative (CHSSI) computational fluid dynamics (CFD)-6, a suite of codes, now called the zonal Navier-Stokes flow (ZNSFLOW) solver, was developed to enable the calculation of aerodynamic problems of Army interest. The suite includes a zonal Navier-Stokes solver and a graphical user interface (GUI) environment for problem setup, interactive visualization, and solver execution. The objectives of the effort were to

1. Develop a scalable version of ZNSFLOW;

2. Add features that would enhance applicability and ease of use;

3. Demonstrate the design utility of the software by solving current Department of Defense (DoD) priority viscous flow problems.

Based on a solver known as F3D (a fully vectorized FORTRAN[1] 77 code), the code was rewritten to provide scalable performance on a variety of architectures. Enhancements included dynamic memory allocation and optimized cache management. Emphasis was placed on user friendliness and ease of use. The distributed interactive computing environment GUI was written to allow some of ZNSFLOW's complex features to be easily employed and to incorporate menu-based help. For example, the solver allows for one-on-one overlaps between grid zones in any direction. The GUI makes the setup for this generalized data exchange intuitive and provides simple error-checking capabilities. Boundary conditions are generalized and can be applied to any surface or line. In addition, the solver can perform computations with a Chimera composite grid discretization technique. The difficulty of using turbulence models with the Chimera technique was overcome by the implementation of a point-wise turbulence model.

ZNSFLOW was designed to operate in both a shared and a distributed memory computer environment. The shared memory version of the solver relies on loop-level parallelism with optimized cache management. The distributed memory version uses the shared memory (SHMEM) library for the Cray T3E and Origin 2000 computers and the message-passing interface (MPI) library for the IBM scalable parallel (SP) computers.

The code has been successful in calculating flow fields, starting with a simple flat plate, flows around missiles at large angles of attack, guided multiple launch rocket system, and the flow field around ten brilliant antitank sub-munitions ejected from an Army tactical missile. Typically, for a grid size of 59 million elements, the performance achieved on a Silicon Graphics, Incorporated, Origin 2000 with 96 processors is 18 times that on a single Cray C-90 processor. Experimental verification of the flow predictions gives confidence in the capability of the code.

[1] Formula Translator

INTENTIONALLY LEFT BLANK

COMMON HIGH PERFORMANCE COMPUTING SOFTWARE SUPPORT INITIATIVE (CHSSI) COMPUTATIONAL FLUID DYNAMICS (CFD)-6 PROJECT FINAL REPORT: ARL BLOCK-STRUCTURED GRIDDING ZONAL NAVIER-STOKES FLOW (ZNSFLOW) SOLVER SOFTWARE

1. INTRODUCTION

The thrust of the work described here is to further develop an existing computational fluid dynamics (CFD) code and make it more accessible for engineers. The code was developed as part of the common high performance computing software support initiative (CHSSI) and is now called the zonal Navier-Stokes flow (ZNSFLOW) solver. ZNSFLOW is actually a suite of codes that basically includes a zonal Navier-Stokes solver and graphical user interface (GUI) environment for problem setup, interactive visualization, and solver execution. The primary goals of the ZNSFLOW CHSSI project are to (a) develop a scalable version of a zonal Navier-Stokes solver, (b) add features to the ZNSFLOW software, which allow general applicability and ease of use, and (c) demonstrate the design utility of the scalable ZNSFLOW software by solving current Department of Defense (DoD) priority viscous flow problems. In keeping with these goals, this report gives a broad overview of the ZNSFLOW CHSSI project, the ZNSFLOW solver and its capabilities, as well as the GUI environment. Some results from test cases are presented to demonstrate recent applications of ZNSFLOW.

2. THE ZNSFLOW SOLVER

The ZNSFLOW solver was originally known as F3D, a fully vectorized (FORTRAN[2]) 77 code used on Cray vector computers such as the C-90.**[1,2]** During the CHSSI program, it has been rewritten to provide scalable performance on a number of computer architectures. Other added enhancements include dynamic memory allocation and highly optimized cache management. Aside from the performance aspects, the solver has been provided with a number of enhancements to make it more user friendly and capable of performing flow field computations for complex configurations of interest to DoD. The solver was written to operate with and without a GUI environment. A large portion of the effort spent on the ZNSFLOW CHSSI project went toward increasing ease of use and general applicability of the ZNSFLOW solver. The distributed interactive computing environment (DICE) GUI allows some of the ZNSFLOW solver's more complex features to be easily employed. For example, the solver allows for one-to-one overlaps between grid zones in any direction. The GUI makes the setup for this generalized data exchange intuitive and provides some simple error-checking capabilities to catch mistakes in creating the input file. Many of the boundary conditions are generalized and can be called for any surface or line. In addition, the solver can perform computations with the Chimera composite grid discretization technique.**[3-5]** By using the Chimera technique, one can greatly simplify the grid topology and grid generation for very complex systems. One of the drawbacks in using the Chimera technique has been the increased complexity and corresponding confusion in applying a turbulence model. A Chimera model can be composed of multiple zones, with each zone

[2] Formula Translator

possibly having a unique grid topology. Most turbulence models have specific directional, orientation, or distance-related requirements for correct application. For a complex Chimera model, applying a turbulence model can be a very complicated process. This problem has been addressed in ZNSFLOW by the installation of a point-wise turbulence model **[6]** that is not orientation specific. This greatly simplifies the setup of the turbulence model. Wall location information is supplied when the wall boundary conditions are set by the user through the GUI. A conventional Baldwin-Lomax turbulence model **[7]** is also available.

2.1 Governing Equations and Solution Technique

The complete set of time-dependent, Reynolds-averaged, thin layer, Navier-Stokes equations is solved numerically to obtain a solution to this problem. The numerical technique used is an implicit, finite difference scheme. Steady state calculations are made to numerically compute the flow field.

2.1.1 *Governing Equations*

The three-dimensional (3-D), time-dependent, generalized geometry, Reynolds-averaged, thin layer, Navier-Stokes equations for general spatial coordinates ξ, η, and ζ can be written as follows **[1]**:

$$\partial_\tau \hat{q} + \partial_\xi \hat{F} + \partial_\eta \hat{G} + \partial_\zeta \hat{H} = Re^{-1} \partial_\zeta \hat{S}, \tag{1}$$

in which

$\xi = \xi(x, y, z, t)$ - longitudinal coordinate;
$\eta = \eta(x, y, z, t)$ - circumferential coordinate;
$\zeta = \zeta(x, y, z, t)$ - nearly normal coordinate; and
$\tau = t$ - time

In Equation (1), $\hat{q}$ contains the dependent variables (density, three velocity components, and energy), and $\hat{F}, \hat{G}$, and $\hat{H}$ are flux vectors. The thin layer approximation is used here, and the viscous terms involving velocity gradients in both the longitudinal and circumferential directions are neglected. The viscous terms are retained in the normal direction, ζ, and are collected into the vector $\hat{S}$. In the wake or the base region, similar viscous terms are also added in the stream-wise direction, ξ. For computation of turbulent flows, the turbulent contributions are supplied through an algebraic eddy viscosity turbulence model developed by Baldwin and Lomax **[7]** or a point-wise turbulence model **[6]**. A technical discussion of the turbulence models is given in Appendix A.

2.1.2 *Numerical Technique*

The implicit, approximately factored scheme for the thin layer, Navier-Stokes equations using central differencing in the η and ζ directions and an upwind scheme in ξ is written in the following **[2]**:

$$\left[I + i_b h \delta_\xi^b (\hat{A}^+)^n + i_b h \delta_\zeta \hat{C}^n - i_b h Re^{-1} \bar{\delta}_\zeta J^{-1} \hat{M}^n J - i_b D_i |_\zeta\right]$$
$$\times \left[I + i_b h \delta_\xi^f (\hat{A}^-)^n + i_b h \delta_\eta \hat{B}^n - i_b D_i |_\eta\right] \Delta \hat{Q}^n$$

(2)

$$= - i_b \Delta t \{\delta_\xi^b [(\hat{F}^+)^n - \hat{F}_\infty^+] + \delta_\xi^f [(\hat{F}^-)^n - \hat{F}_\infty^-] + \delta_\eta (\hat{G}^n - \hat{G}_\infty)$$
$$+ \delta_\zeta (\hat{H}^n - \hat{H}_\infty) - Re^{-1} \bar{\delta}_\zeta (\hat{S}^n - \hat{S}_\infty)\} - i_b D_e (\hat{Q}^n - \hat{Q}_\infty),$$

in which h = Δt or (Δt)/2. The free-stream fluxes are subtracted from the governing equation to reduce the possibility of error from the free-stream solution corrupting the converged solution. Here, δ is typically a three-point, second order, accurate central difference operator; $\bar{\delta}$ is a midpoint operator used with the viscous terms; and the operators δ_ξ^b and δ_ξ^f are backward and forward three-point difference operators. The flux $\hat{F}$ has been eigensplit and the matrices $\hat{A}, \hat{B}, \hat{C}$, and $\hat{M}$ result from local linearization of the fluxes about the previous time level. Here, J denotes the Jacobian of the coordinate transformation. Dissipation operators D_e and D_i are used in the central space differencing directions. The smoothing terms used in the present study are of the form

$$D_e |_\eta = (\Delta t)\, J^{-1} \left[\varepsilon_2 \bar{\delta} \rho(B) \beta \bar{\delta} + \varepsilon_4 \bar{\delta} \frac{\rho(B)}{1+\beta} \bar{\delta}^3 \right] |_\eta J,$$

and

$$D_i |_\eta = (\Delta t)\, J^{-1} \left[\varepsilon_2 \bar{\delta} \rho(B) \beta \bar{\delta} + 2.5 \varepsilon_4 \bar{\delta} \rho(B) \bar{\delta} \right] |_\eta J,$$

in which

$$\beta = \frac{|\bar{\delta}^2 P|}{|(1+\delta^2) P|},$$

and ρ(B) is the true spectral radius of B. The idea here is that the fourth difference will be adjusted downward near shocks (e.g., as β gets large, the weight on the fourth difference drops while the second difference adjusts upward).

2.2 Chimera Composite Grid Scheme

The Chimera overset grid technique greatly adds to the number of applications to which the ZNSFLOW solver can be applied. Although the ZNSFLOW solver can be applied to computational meshes with Chimera topology, note that the software used to create such computational meshes is not included with the ZNSFLOW suite of software. The Chimera over-set grid technique, which is ideally suited to multi-body problems,**[8-10]** involves generating independent grids about each body and then over-setting them onto a base grid to form the complete model. This procedure reduces a complex multi-body problem into a number of simpler sub-problems. An advantage of the over-set grid technique is that it allows computational grids to be obtained for each body component separately and thus makes the grid generation process easier. Because each component grid is generated independently, portions of one grid may

lie within a solid boundary contained within another grid. Such points lie outside the computational domain and are excluded from the solution process. Equation (2) has been modified for Chimera over-set grids by the introduction of the flag i_b to achieve just that. This i_b array accommodates the possibility of having arbitrary holes in the grid. The i_b array is defined so that $i_b = 1$ at normal grid points and $i_b = 0$ at hole points. Thus, when $i_b = 1$, Equation (2) becomes the standard scheme, but when $i_b = 0$, the algorithm reduces to $\Delta\hat{Q}^n = 0$ or $\hat{Q}^{n+1} = \hat{Q}^n$, leaving $\hat{Q}$ unchanged at hole points. The set of grid points that forms the border between the hole points and the normal field points is called inter-grid boundary points. These points are revised by interpolating the solution from the overset grid that created the hole. Values of the i_b array and the interpolation coefficients needed for this revision are provided by a separate algorithm.[3]

Figure 1 shows an example in which the parent missile grid is a major grid, and the brilliant anti-armor (BAT) sub-munition grid is a minor grid. The sub-munition grid is completely overlapped by the missile grid, and thus, its outer boundary can obtain information by interpolation from the missile grid. Similar data transfer or communication is needed from the sub-munition grid to the missile grid. However, a natural outer boundary that overlaps the sub-munition grid does not exist for the missile grid. The over-set grid technique creates an artificial boundary or a hole boundary within the missile grid, which provides the required path for information transfer from the sub-munition grid to the missile grid. The resulting hole region is excluded from the flow field solution in the missile grid.

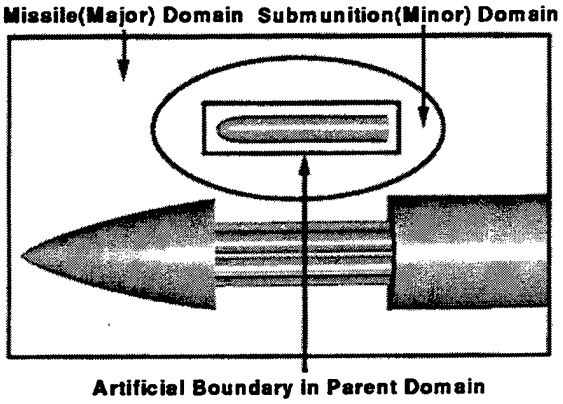


Figure 1. Inter-grid Communications.

2.3 Distributed and Shared Memory ZNSFLOW

The ZNSFLOW software has been targeted to operate on both shared memory and distributed memory architectures. In order to fulfill the CHSSI program requirement that the ZNSFLOW solver be scalable on applicable computers, it was decided to create two versions of the ZNSFLOW solver, with one version optimized to operate on shared memory architectures and the other optimized to operate on distributed memory architectures. The shared memory version of the solver employs loop-level parallelism that has highly optimized cache management. The distributed memory version is not as fully developed as the shared memory version and is currently not able to

perform computations using the Chimera scheme. Once the ZNSFLOW software is complete, the differences between the multiple versions of the solver should be transparent to the user. Both versions of the ZNSFLOW solver apply the same unsteady Reynolds-averaged thin layer Navier-Stokes equations, as described previously, to compute flow field solutions with no changes in the time-tested solution algorithm. However, the versions use different programming techniques to achieve scalable performance for their intended computer architectures. The immediately following sections (2.3.1 and 2.3.2) describe some of the technical aspects and performance of the distributed and shared memory versions of the ZNSFLOW solver.

2.3.1 *Shared Memory ZNSFLOW*

Many modern parallel computers are now based on high-performance reduced instruction set computing (RISC) processors. The shared memory version of ZNSFLOW is written to perform efficiently on these computers. The key breakthrough in determining a methodology for optimizing and parallelizing the ZNSFLOW solver was the realization that many of the new systems seem to lend themselves to the use of loop-level parallelism. This strategy offers the promise of allowing the solver to be parallelized with absolutely no changes in the algorithm. Note that it is difficult to efficiently use loop-level parallelism on anything but a shared memory architecture and only recently have vendors started shipping shared memory architectures that are based on RISC processors with aggregate peak speeds exceeding a few Gflops[3]. Parallel high performance computers (HPCs) often employ approximately 100 RISC processors. With the speed of RISC processors, it may not be necessary to use more than 100 processors to meet most users' needs. However, for this assumption to be true, a reasonable percentage of the peak processing speed of each processor must be used. Programming for the use of a limited number of powerful processors (e.g., 10 to 100 processors) has some advantages over programming for a computer that employs approximately 1,000 relatively weaker processors. Using significantly fewer processors can

1. Allow the use of parallelization techniques that may support only a limited degree of parallelism;

2. Decrease the extent to which the parallel efficiency of the algorithm is degraded;

3. Decrease the percentage of the run time spent passing messages; and

4. Decrease the effect of Amdahl's Law.[**11**]

Several methods were used to improve code performance for computers employing multiple RISC processors. The goal was to achieve both serial and parallel efficiency. To achieve high serial efficiency on a RISC processor, the programmer must be mindful of the cache miss rate and the translation "look-aside" buffer (TLB) miss rate. Some of the programming techniques used to accomplish this are also beneficial to the program's parallel performance. Some of the programming techniques used to optimize code performance are described next.

[3] One billion floating point operations per second

• Indices of arrays were reordered to improve spatial and temporal memory access locality. For example, if there is a long complex loop that employs values associated with a single data point, then it is more efficient to store those values in array Q(N,J,K,L) than in array Q(J,K,L,N) in which N is some small integer such as 5 or 6.

• Multiple arrays used as a group were merged. For example, if the arrays XX(J,K,L), XY(J,K,L), and XZ(J,K,L) are needed for the same equation, they should be merged to form the single array XXYZ(3,J,K,L).

• Loops in nested loops were reordered. For example, if one has a nested loop such as

```
DO ... M=1,5
 DO ... N=1,5
  DO ... L=1,LMAX
   DO ... K=1,KMAX
    Several lines of code involving arrays such as A(K,L,N,M).
```

It will probably produce far fewer cache and TLB misses if the loop nest can be rewritten as

```
DO ... L=1,LMAX
 DO ... K=1,KMAX
  DO ... M=1,5
   DO ... N=1,5
    Several lines of code that now involve arrays such as A(N,M,K,L).
```

This will have the added benefit of potentially supporting more aggressive forms of loop unrolling.

• Matrix transposition operations for invariant/relatively invariant arrays were kept in memory between uses and were revised only when needed.

• Loops were sized by data requirements to allow sets of operations to be performed so that the data used were "locked" into the cache memory. This technique is not very beneficial for small cache sizes. For good efficiency, an "off-chip" cache size of at least 1 megabyte (MB) is needed. Properly sizing the loops virtually eliminated cache misses associated with scratch arrays.

Note that the sample computer code and variables are for the FORTRAN language in which most of the solver is written. If a different computer language, such as C is used, the array indices and loops may need to be ordered differently for peak efficiency. For a more formal discussion of how and why the above programming techniques are used to improve code performance, refer to Appendix B and Pressel.**[12]**

A generic missile configuration was used for many of the tests of the parallelized code. In these tests, a 1,000,000-point grid (see Figure 2) was used to check the accuracy of the results. The computed results obtained with the parallelized code were compared with those obtained using the vectorized code on a Cray C-90. These computed results were compared with the experimental data obtained from the Defense Evaluation and Research Agency (DERA), United Kingdom, for the same configuration and test conditions.**[13,14]** For this case, the computation on the C-90

used 18 mega-words (144 MB) of memory and approximately 7.5 hours of central processing unit (CPU) time. Once the accuracy of the computed results was verified, performance studies were conducted for grid sizes ranging from 1 to 59 million grid points. Figure 3 shows the circumferential surface pressure coefficient distribution of the missile at a selected longitudinal station.**[14]** Computed results from both vectorized (C-90) as well as the parallelized versions of the code are shown to lie on top of one another and are thus in excellent agreement.

Results were obtained by using a highly efficient serial algorithm as the starting point and taking great care not to make any changes in the algorithm. Initial efforts to run the vector-optimized version of this code on one processor of a Silicon Graphics, Incorporated (SGI) Power Challenge (75-MHz R8000 processor) proved to be extremely disappointing. After aggressively tuning the code for a low-cache miss rate and good pipeline efficiency, the authors achieved a factor of 10 improvement in the serial performance of this code. At this point, the percentage of peak performance from the RISC-tuned code using one processor on the SGI Power Challenge was the same as the vector-tuned code on one processor of a Cray C-90.

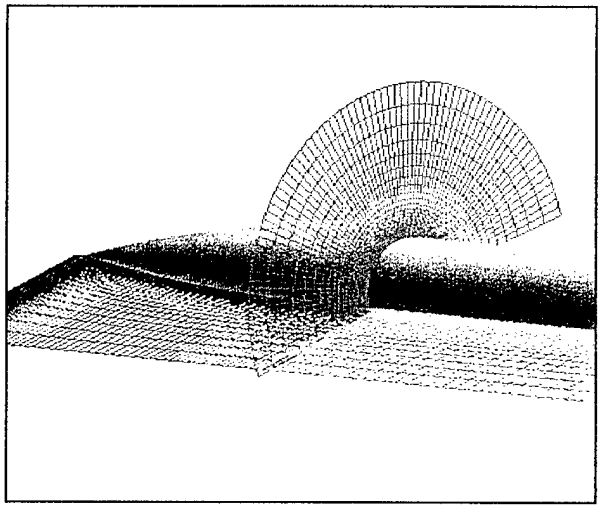


Figure 2. 1-Million-Point Key Technical Area (KTA) Computational Grid.

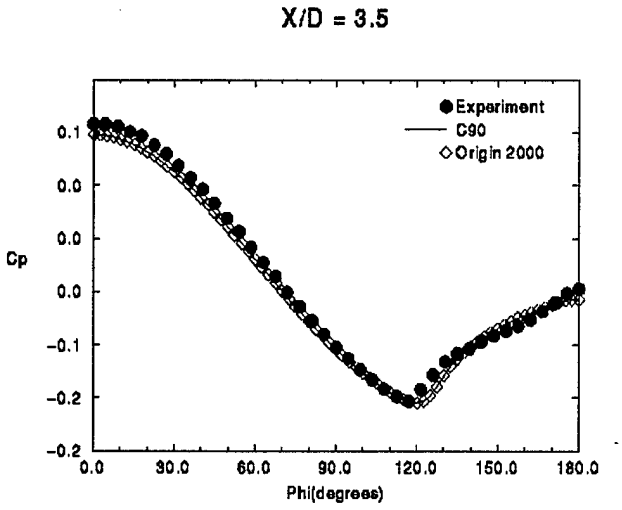


Figure 3. Pressure Coefficient Comparison. **[13,14]**

A key factor was the observation that processors with a large external cache (e.g., 1 to 4 MB in size) could enable the use of optimization strategies that simply were not possible on machines such as the Cray T3D and Intel Paragon which only have 16 kilobytes of cache per processor. This relates to the ability to "size" scratch arrays so that they will fit entirely in the large external cache. This can reduce the rate of cache misses associated with these arrays, which go all the way back to main memory, to less than 0.1% (the comparable cache miss rates for machines such as the Cray T3D and Intel Paragon could easily be as high as 25%). The immediately preceding data (serial performance and cache miss rate) were obtained from a highly optimized version of the F3D code. ZNSFLOW uses most of the solver coding from this optimized FORTRAN-only version of F3D. However, the shared memory version of ZNSFLOW also employs C language coding primarily to control the main integration loop of the solver, allocate dynamic memory, and

communicate with DICE software. Thus, there are some differences in the performance of ZNSFLOW and the previously mentioned optimized version of F3D. Figure 4 shows a graphical comparison of the "speedup" for the latest version of ZNSFLOW running on an SGI Origin 2000 (300-MHz R12000 processor) versus the vector version of the original F3D code running on a single Cray C-90 processor. The data graphed in Figure 4 are presented in Table 1. Note that the SGI Origin execution time for the 1- and 59-million-point cases was acquired through actual time measurement of ZNSFLOW execution. However, only the 1-million-point case was executed on the Cray C-90. The performance of the 59-million-point case for a single Cray C-90 processor was estimated. Figure 5 displays ZNSFLOW performance graphs for 1-, 10-, and 59-million grid point KTA data sets.

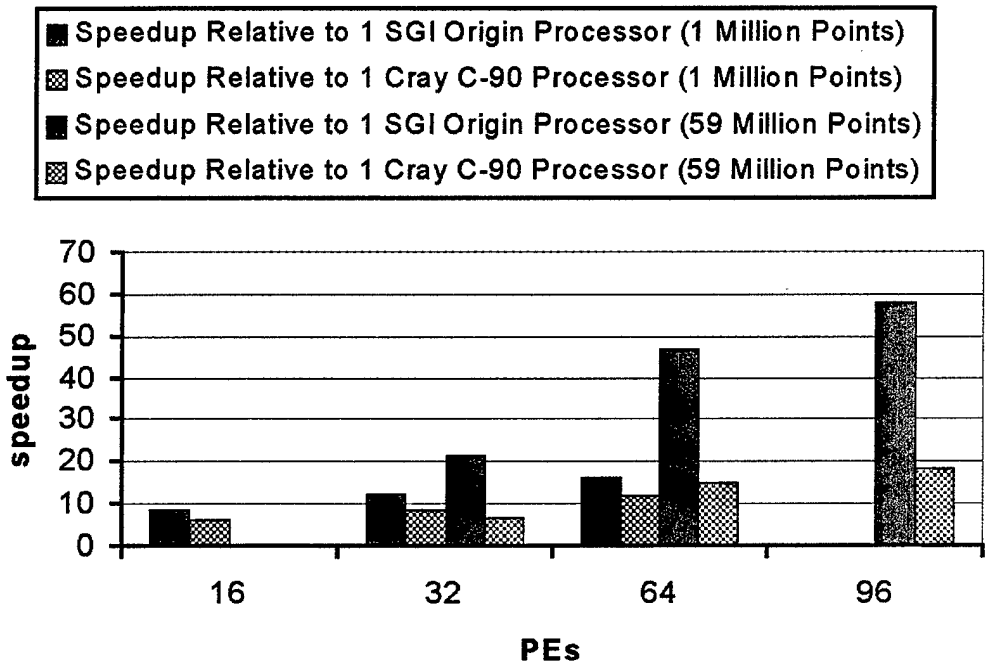
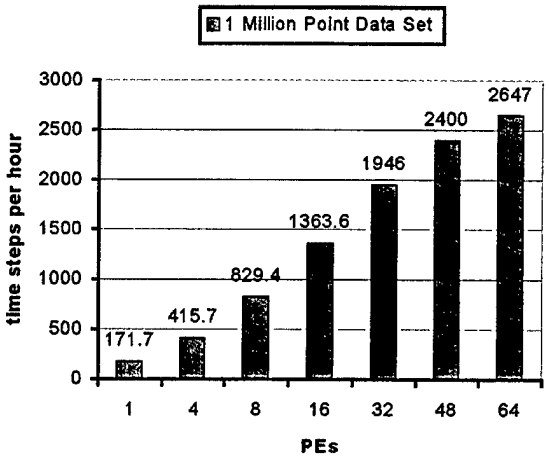


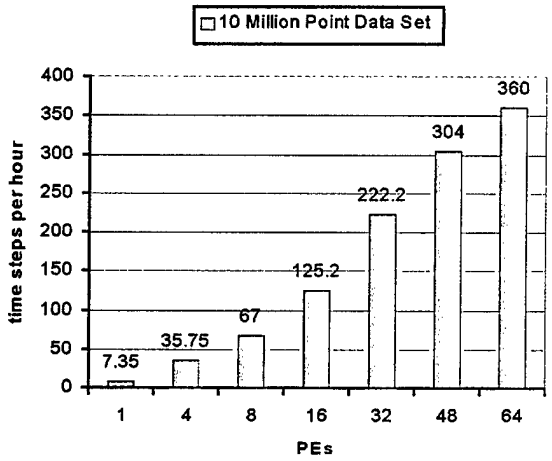
Figure 4. Graph of KTA Timing Data Speedup.

Table 1. KTA Timing Data Speedup

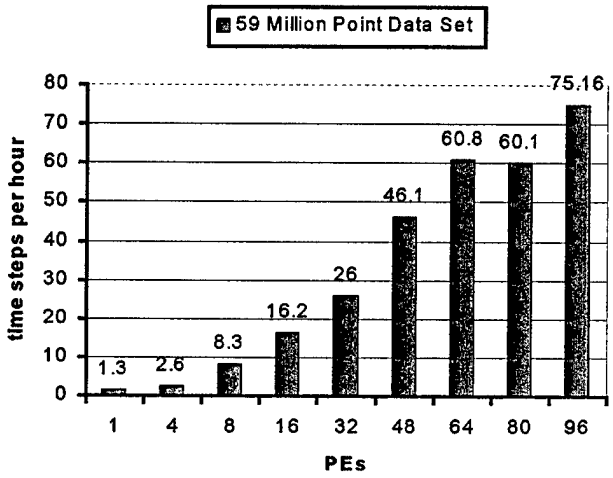
Grid Size (in millions of points)	Number of Processors	Speedup Relative to One Origin 2000 Processor	Speedup Relative to One C-90 Processor
1	16	8.3	6.0
1	32	12.0	8.6
1	64	16.0	11.6
59	32	21.5	6.3
59	64	46.8	14.8
59	96	57.8	18.3



5a. Performance for 1-million-point data set.



5b. Performance for 10-million-point data set.



5c. Performance for 59-million-point data set.

Figure 5. Performance Results for 1-, 10-, and 59-Million Grid Point KTA Data Sets.

While linear speedup is desired, it is generally impossible to obtain linear speedup using loop-level parallelism. The best that can be achieved is a curve with a staircase effect. Because of the limited number of graphed computer timings in Figure 5, the staircase effect is only evident in one graph, the 59-million-point data set timings shown in Figure 5c. For 64 and 80 processors, the number of time steps per hour is nearly identical. A performance increase is shown when the number of processors is raised to 96. The source of this effect is the limited parallelism (especially when working with 3-D codes) associated with loop-level parallelism and is the basic result of integer division. Table 2 demonstrates this effect. Using the data in Table 2 as an example, one can see that the time to complete a case remains the same when 8 to 14 processors are used. When 15 processors are used, the computation will theoretically require only half the time of a computation using 14 processors. From a different perspective, the data in Table 2 show that the amount of time the computation requires for completion is the same for 8 to 14 processors.

Therefore, when one is given the option of using 8 to 14 processors, theoretically, the most efficient number of processors to use is 8. These points should be kept in mind to aid in determining efficient numbers of processors for use in actual computations.

Table 2. Predicted Speedup for a Loop With 15 Units of Parallelism

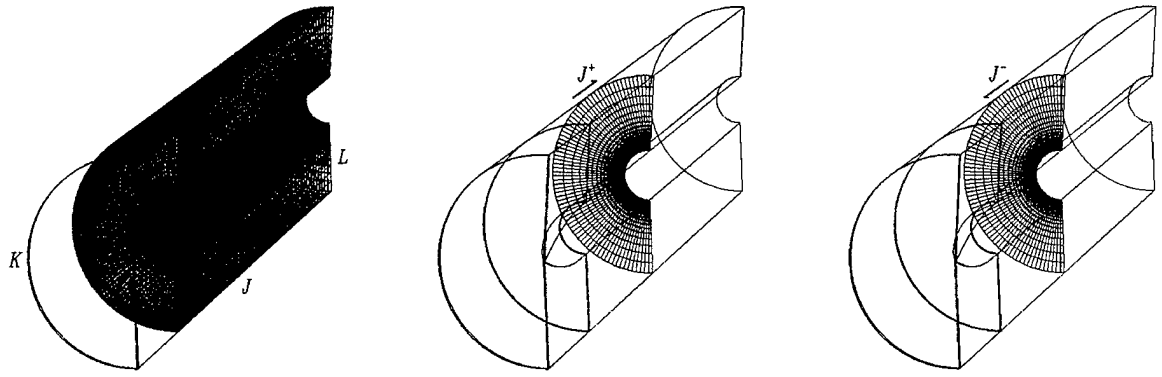
Number of processors	Maximum Units of Parallelism Assigned to a Single Processor	Predicted Loop-Level Parallelism Speedup
1	15	1.000
2	8	1.875
3	5	3.000
4	4	3.750
5 to 7	3	5.000
8 to 14	2	7.500
15	1	15.000

2.3.2 *Distributed Memory ZNSFLOW*

The ZNSFLOW distributed memory code is the message-passing implementation of ZNSFLOW. To better explain the parallelization issues, an overview of the typical ZNSFLOW computation steps is given next. All operations proceed again on a zone-by-zone basis, with inactive zone data stored either in memory or on a fast mass storage device. A single zone is constructed of a regular *NJ* x *NK* x *NL* block of cells aligned with *J*, *K*, and *L* directions. The *J* direction is assumed to be stream wise and is treated semi-implicitly with two solver sweeps in the J^+ and J^- directions. During the J^+ sweep, for each consecutive stream-wise plane, the grid points are coupled in the *L* direction, while they are treated independently in the *K* direction. This requires a solution of *K* tri-diagonal systems of size *L* with 5 x 5 blocks. In the J^- sweep, the roles are reversed, with the coupling present in *K* direction only and *L* block-tri-diagonal systems of size *K*. Before the sweeping can commence, a volume calculation of the right-hand side (RHS) must take place (see Figure 6). An efficient parallel implementation of these two distinct computation phases, RHS formation, and solver sweeps, is crucial to the overall effectiveness and scalability of the code.

Between the two computation-intensive stages of the code, the RHS formation yields itself to parallelization most easily. This is a volume computation, in which each grid point is operated independently, with only older values at neighboring points being required to complete the computation. The entire set of zone cells can be distributed over the available processing elements (PEs) in an arbitrary manner. However, for the sake of subsequent solver computations, it makes sense to decompose only *K* and *L* grid dimensions, leaving an entire *J* dimension associated with a single PE. The *K*-*L* plane is mapped onto a rectangular grid of all PEs. To

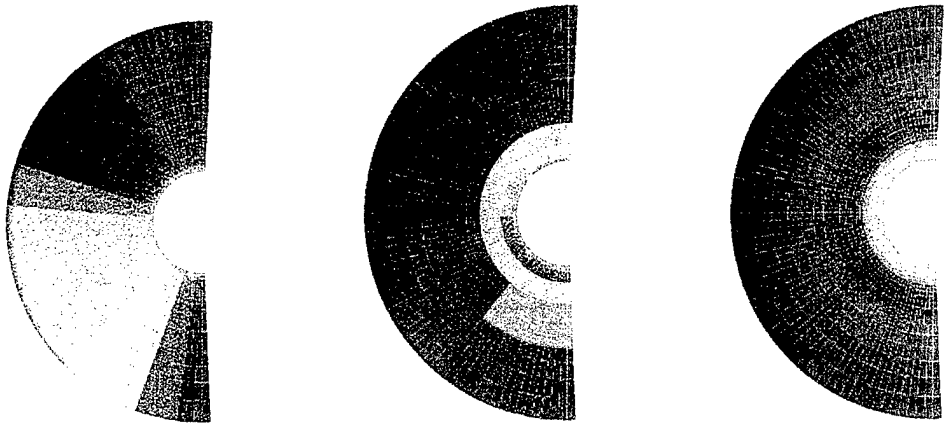
avoid repetition of inter-processor transfers, each rectangular portion of the *K-L* plane also contains two layers of "ghost" points that track the two closest sets of values in the sub-grids belonging to neighboring PEs.



6a. RHS volume computation 6b. J^+ sweep 6c. J^- sweep

Figure 6. Data Orientation and Activity During ZNSFLOW Phases for the Last Zone of the Benchmark Problem.

The parallelization of the solver sweeps is not as straightforward. The algorithm requires sequential processing in the *J* direction and can also be simultaneously parallelized in both *K-L* directions, only at a greatly added computational cost, e.g., via a cyclic reduction algorithm. An alternate method is to accept serial treatment of the *J* and *L* directions (*J* and *K* for J^- sweep) and to devote all PEs to parallelizing the *K* dimension (*L* for J^- sweep). This approach has an obvious disadvantage, since the scalability is not maintained as the number of PEs exceeds either the *NK* or *NL* zone dimensions. In typical computations, however, the number of PEs and the zone dimensions are matched so that the problem does not arise. Therefore, for the solver sweeps, the desired data distribution has the entire *J* and *L* dimensions associated with a single PE, and the *K* dimension is decomposed among all available PEs for the J^+ sweep. *J* and *K* dimensions are associated with a single PE and the *L* dimension distributed for the J^- sweep. This requires repeated reshaping of a small number of arrays between the original and two solver-specific layouts (see Figure 7). A number of smaller parallelization issues had to be resolved as well, including parameter reading and broadcasting among PEs, efficient disk input/output (I/O), and exchange of boundary data between zones.



7a. J^+ sweep 7b. RHS volume computation 7c. J^- sweep

Figure 7. Data Distribution During ZNSFLOW Phases.

The initial attempt to port the ZNSFLOW code to a scalable architecture employed the Cray T3D and CRAFT (not an acronym) shared memory programming model. The advantages of code maintainability and ease of transition were offset by the poor performance, and alternate approaches were explored. The more difficult task of rewriting the code in a message-passing framework was undertaken, and the parallel virtual machine (PVM)-based code provided initial speedups. The reshaping of the arrays during solver sweeps was, however, a difficult target for efficient implementation when two-sided PVM communication was used. A much better solution was found in the form of the one-sided shared memory (SHMEM) Cray communication libraries. In addition to eliminating concerns about deadlocking, the use of SHMEM reduces message latency and increases bandwidth. Apart from the communication issues, some scalar optimization of the code was attempted in order to extract a reasonable fraction of peak speed on cache-constrained architectures, but that aspect still leaves something to be desired. A variation based on the message-passing interface (MPI) library has since been added to the code base in order to ensure portability to platforms that do not support SHMEM, such as IBM SP and the Sun HPC. It is anticipated that both the SHMEM and MPI portions will be replaced with a single one-sided MPI-2 version as this standard becomes widely accepted.

Speed and scalability of the message-passing code is tested on three architectures, using Mach 1.8 flow past an ogive cylinder at a 14° angle of attack on a three-zone 1-million-point coarse grid, and the same geometry at Mach 2.5 on a 10-million-point fine grid. The results are listed in terms of time steps per hour in Tables 3 and 4 and are shown in graphical form in Figures 8 and 9. The Cray T3E and SGI Origin platforms use the SHMEM-based version of ZNSFLOW, while the IBM SP employs the less efficient MPI-based version. For comparison, the Cray C-90 version of the code achieved 227 time steps per hour for the 1-million-point case. As expected, the plots show better scalability for the refined grid than for the coarse one, as parts of the current implicit solver contain parallelism only of the order of K or L dimensions. These dimensions are 75 and 70, respectively, for the coarse grid, and 180 and 140 for the refined one.

The graphs exhibit visible notches around 70 and 75 PEs for the coarse grid and around 70 PEs for the fine grid; these are thresholds at which the integer number of points per PE (for the loops with K or L parallelism) decreases by one. A number of predictable secondary gradients in performance occur as the integer number of points per PE changes for the K-L layouts. A sample Mach number field at the conclusion of the 1-million-point simulation is shown in Figure 10.

Table 3. Scalable Performance of Distributed Memory ZNSFLOW Solver on Several Platforms in Time Steps per Hour for the 1-Million-Point Case

PEs	T3E-1200	O2K (300 MHz)	SP (160 MHz)
8	349	382	199
16	616	618	288
24	888	838	335
32	1062	882	342
40	1324	989	374
48	1431	1083	420
56	1642	1161	428
64	1705	1050	423
72	2141	1326	405
80	2280	1382	420
88	2443	1320	396
96	2478		
104	2673		
112	2711		
120	2914		
128	2948		

Table 4. Scalable Performance of Distributed Memory ZNSFLOW Solver on Several Platforms in Time Steps per Hour for the 10-Million-Point Case

PEs	T3E-1200	O2K (300 MHz)	SP (160 MHz)
16	70		41
24	99	84	54
32	127	97	62
40	152	113	72
48	179	142	81
56	190	134	84
64	203	133	89
72	248	158	93
80	247	157	94
88	276	153	95
96	298		
104	317		
112	337		
120	355		
128	327		

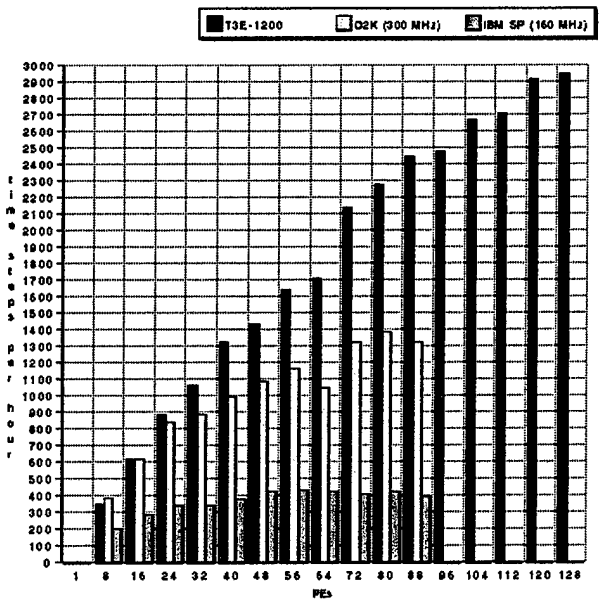


Figure 8. Graph of Scalable Performance of Distributed Memory ZNSFLOW Solver on Several Platforms for the 1-Million-Point Benchmark Case.

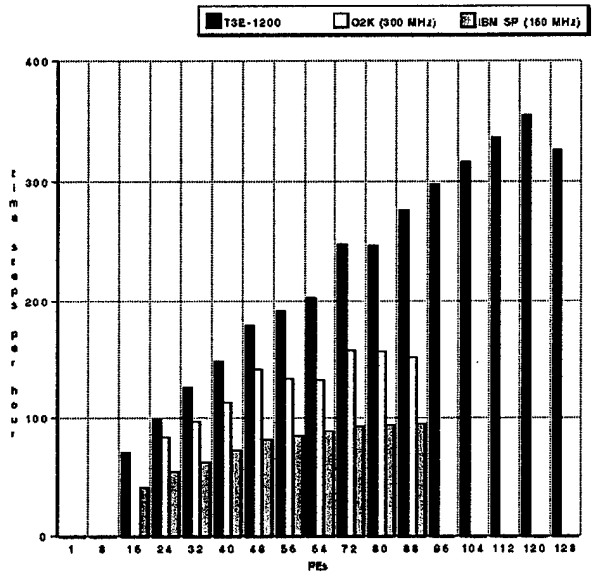


Figure 9. Graph of Scalable Performance of Distributed Memory ZNSFLOW on Several Platforms for the 10-Million-Point Case.

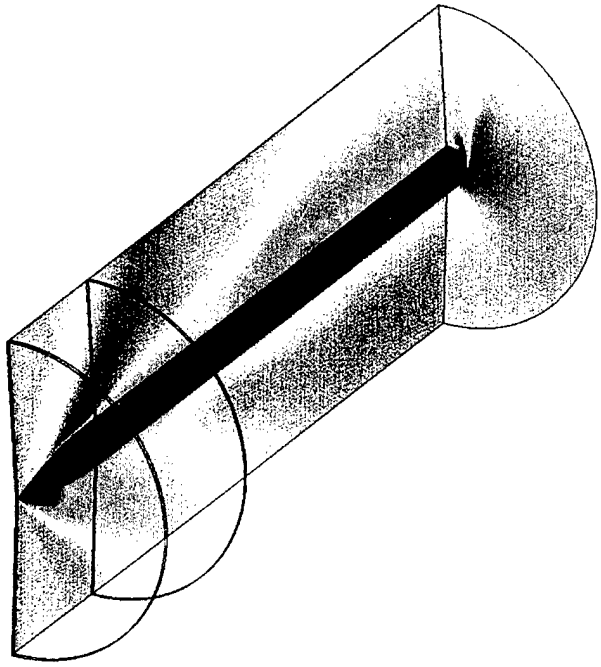


Figure 10. Ogive Cylinder: Mach Number Contours for the 1-Million-Point Case.

2.4 ZNSFLOW Validation

The predecessor of ZNSFLOW, F3D, has a long track record of providing accurate CFD computations. There are numerous reports that document CFD results obtained with this solver, and some are listed in the reference section of this report. Some sample cases are provided to users of ZNSFLOW when they obtain the code. These cases are simple, classic CFD computations that demonstrate the accuracy of the code and show how to correctly apply the ZNSFLOW solver. Each validation case is provided with experimental data for comparison. Validation cases are completely documented within the yet-to-be formally published ZNSFLOW user manual entitled, "Documentation and User's Guide for the ZNSFLOW Code".**[15]** A copy of the ZNSFLOW user manual is provided when the ZNSFLOW software is obtained. The ZNSFLOW user manual may also be downloaded from the ZNSFLOW CHSSI web site at http://www.arl.hpc.mil/chssi/cfd6/.

ZNSFLOW software has also been reviewed by the DoD high performance computing modernization office CFD computational technical area (CTA) lead, Jay Boris, during required CHSSI software tests in May 1998. ZNSFLOW met or surpassed scalable speedup criteria for available computers, and the validation cases provided documentation of the solver's computational accuracy.

3. THE DISTRIBUTED INTERACTIVE COMPUTING ENVIRONMENT (DICE)

As stated earlier, ZNSFLOW is a suite of codes. Part of that suite is DICE.**[16]** DICE provides a GUI that allows a user to create an input file for the ZNSFLOW solver. DICE can also be used to execute the ZNSFLOW solver once the input file has been created. In addition, once the solver is executing, DICE can provide real-time visualization of the flow field as it is being computed. Even if the execution of the solver were initiated from a previous day, DICE would allow the user to connect to the application on a remote computer and visually monitor its progress on a local workstation. DICE provides a number of options for visualizing data. The user can choose surface contours, iso-surfaces, x-y plots, or spreadsheets to display the data. At present, only the shared memory solver has been integrated into DICE. However, DICE can still be used to create the input file and perform visualization of solution files when it is used with the distributed memory solver.

Figure 11 shows some of the GUI windows that a user may access. A flow field visualization window is visible. The user may interactively rotate or translate the object in the window to view the flow field from any position. Beneath the visualization window is the boundary condition setup window. To the right of the boundary condition window is a data directory window. This allows the user to drag and drop specific solver-generated data to DICE utilities such as the iso-surface plotter. Farther to the right is the solver execution window. More controls for executing the solver on multiple platforms are available. To the right of the visualization window is the main interface from which all the other windows are initiated.

It is important to note that DICE is not only a GUI but an environment that includes a heterogeneous distributed memory system called network distributed global memory (NDGM).**[17]** NDGM uses a client-server approach that allows separate distributed applications to access a single contiguous data buffer that may span the memory of several computers. This system forms the bottom layer of the DICE data hub. The hierarchical data format (HDF) from the National Center for Supercomputing Applications (NCSA) serves as a data organization layer above NDGM. HDF4 has been modified to allow data sets to exist on disk, in NDGM, or in both. For example, a static grid could be stored on disk for local access, while calculated scalars could be stored in an NDGM buffer that is revised at every computational iteration. A convenient interface layer sits above HDF and provides consistent access to both structured and unstructured data as well as groups of data sets. This layer contains both tool command language (TCL) and C programming language application programmers' interfaces (APIs). Together, these three layers comprise the data hub in DICE and are known as the DICE data directory. Direct access to the DICE data directory by a code is accomplished via the DICE application interface (DAI). Several heavily used codes have been outfitted with DAI calls to allow run time visualization. The DICE data directory has proved extremely useful as a common data rendezvous for codes executing on HPCs and visualization.

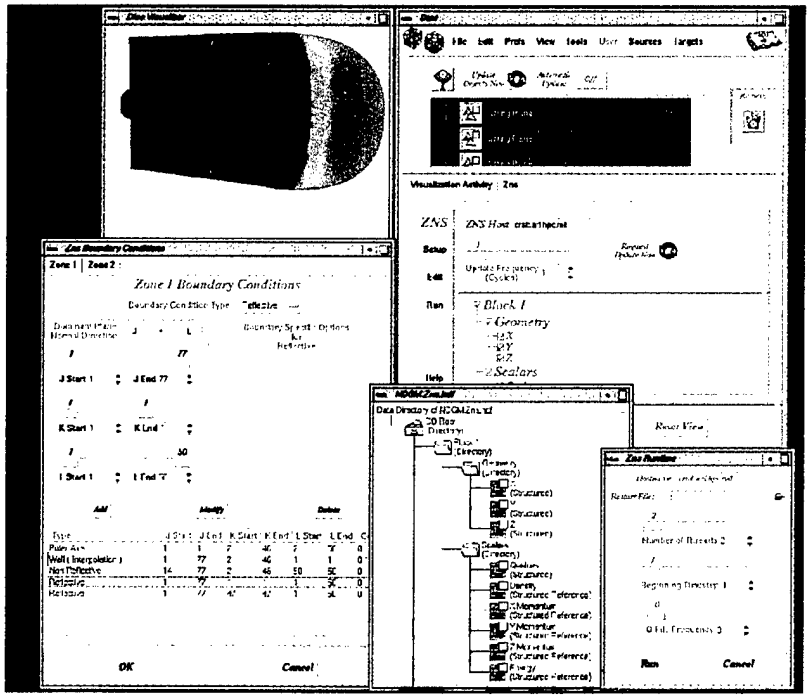


Figure 11. DICE GUI Windows.

NDGM provides DICE with a physically distributed, logically shared, unstructured memory buffer. Instead of handling the mapping and un-mapping of memory pages automatically, NDGM is accessed through a subroutine interface. While less automatic, this allows applications to form a "cooperative shared memory" that is simple yet efficient. NDGM is a client-server system that consists of multiple server processes and an API for clients. Each server maintains a section of a virtual contiguous buffer and field requests for data transfer and program synchronization. Clients use the API to transfer data in and out of the virtual buffer and to coordinate their activity. Calls to the API result in lower level messages being sent to the appropriate NDGM server which keeps track of its piece of the total virtual buffer. The API translates the global memory address into a local address that the server then transfers from its local memory.

Client programs use the API to access the virtual NDGM buffer as contiguous bytes. No structure is placed upon the NDGM buffer; the application can impose any structure on this buffer that is convenient. In addition, NDGM is designed to implement a system of applications in contrast to a single monolithic parallel application. The API includes facilities to get and put contiguous memory areas, get and put vectors of data, acquire and release semaphores, and to initialize and check into multiple barriers. For synchronization purposes, the API provides barriers and semaphores. Checking into a barrier will result in the process blocking until the barrier value reaches zero. Requesting a semaphore will block until the client who currently owns the requested semaphore releases it.

The NDGM server process handles all requests for data transfer and synchronization. This is a stand-alone program that waits for new connections from clients and services their requests. Each server maintains a local memory buffer that maps into the virtual buffer address space. This local buffer can be in one of three locations: local address space (obtained via the "malloc" command), system shared memory, or a local file. If system shared memory is used, a client executing on the same physical machine as the server accesses the shared memory instead of making requests to a server. This access is transparent to the NDGM client application and results in faster data transfers. Using a file as the server's local storage allows NDGM servers to restart with their local memory already initialized.

Clients and servers run on top of a layered MPI. Similar in concept to well-known message-passing interfaces such as PVM or MPI, this layer provides a level of abstraction, freeing the upper layers from the details of reading and writing data. The NDGM message-passing layer has fewer facilities than either PVM or MPI but is designed to pass NDGM data efficiently with minimal copying. This layer provides calls to establish connections, send messages, probe for incoming messages, read messages, and close connections.

The actual inter-process data transfer is accomplished by the drivers. Current drivers include transport control protocol/internet protocol (TCP/IP) sockets, PVM, and first in-first out (FIFO). Each driver has functions to open as a client or server, read, write, and probe for incoming messages. When possible, each driver also implements a "select" function to monitor several open connections. A single NDGM system can mix nodes that use different drivers.

NDGM has been used to develop parallel applications, but it is particularly useful as a "data rendezvous" for a collection of applications. A parallel computationally intensive code can write a snapshot of data to NDGM and then continue its processing. The data can then be visually inspected through 2-D plots and 3-D surfaces, but they do not inhibit the progress of the code. NDGM provides a distributed, heterogeneous unstructured buffer. To provide some structure to this buffer, DICE uses HDF, a well-known and widely used format. HDF is designed to allow an orderly access to structured and unstructured data sets. All access is accomplished through a well-defined application programmer's interface. HDF is designed to access data via disk files. DICE alters some of the low level access routines to allow HDF to access NDGM as well as disk files.

HDF defines a full-featured data format for structured and unstructured data sets as well as groups of data. It does not place restrictions on the organization of these data sets. To simplify access, DICE adds a convenience layer on top of HDF, which has been previously mentioned—the DICE data directory (DDD). Modeled after the UNIX™ file system, DDD provides facilities for mounting data sets and making subdirectories to help organize complex data. DDD provides for structured data sets, unstructured data sets, and directories. In addition, DDD provides a "reference" data set that points to a subsection of previously defined data. In this fashion, a single data file can reside on disk, in memory, or in both and can contain many different types of data.

Through the use of NDGM, HDF, and DDD, the data organization of DICE provides a level of abstraction for enormous distributed data sets. Computational code, visualization, and user interface can all interact with the data in a well-defined method without severely limiting performance. Since all the sections are modular, portions of the data abstraction can be

physically located to optimize the whole application's usability. DICE has proved to be an exceptional computational environment for high performance computing software and is currently used to support several codes developed under CHSSI in different CTAs.

4. ZNSFLOW DEMONSTRATION CASES

Demonstration cases were chosen to show the capabilities of the ZNSFLOW software. Both of the demonstration problems require viscous Navier-Stokes CFD modeling for accurate flow field solutions. Two demonstration cases were run on an SGI Origin 2000 computer. The first of the two cases to be discussed is the guided multiple launch rocket system (MLRS) missile. The guided MLRS computational model is built to answer questions about the use of canards to perform controlled maneuvers for a missile with wraparound tail fins. A second demonstration case shows the capability of ZNSFLOW to model complex multi-body systems. Computational models were built for computing the flow field around 10 BAT sub-munitions being ejected from an Army tactical missile (ATACM). The complexity and uniqueness of this type of multi-body problem result from the aerodynamic interference of the individual components, which include 3-D shock-shock interactions, shock-boundary layer interactions, and highly viscous-dominated separated flow regions.

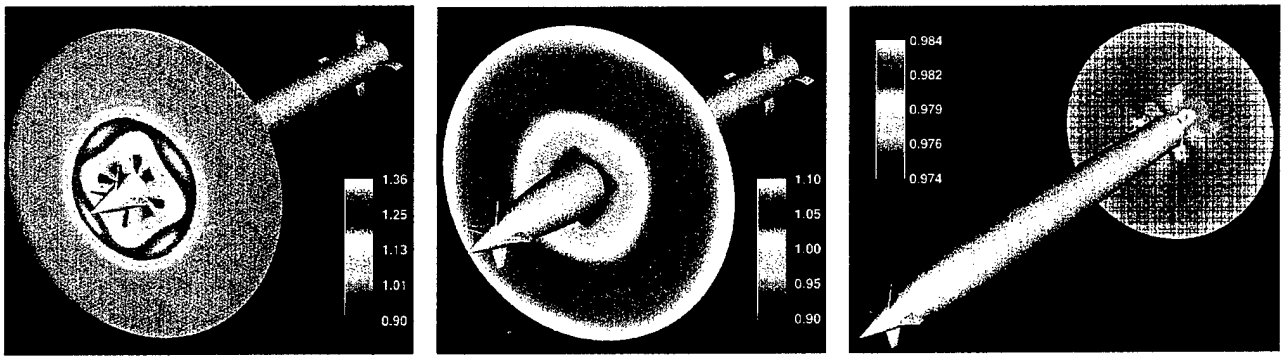
4.1 Computations for the Guided MLRS Missile

The computations will hopefully provide insight for engineers into the interaction of canard-induced flow field disturbances with the down-stream wraparound tail fins. Providing control for a missile with wraparound tail fins is more complex than with normal tail fins. The curvature of the wraparound fins allows for easy storage because the fins fold against the missile body while in the launch tube. Immediately after launch, the fins unfold to stabilize the missile. The cylindrical shape of the wraparound fin is advantageous for packaging, but it can also compromise the dynamic stability of the missile. Wraparound fins have a number of unique aerodynamic traits, the most infamous of which is the roll moment that they generate; this may change in sign and magnitude during the course of a trajectory. The roll moment contributes to the missile spin rate. During the course of flight of a wraparound fin missile, it is possible for its spin rate to increase or decrease more than once. In addition, the direction of spin may change. This type of behavior can produce poor flight dynamics. CFD can be a useful tool for predicting the aerodynamics of wraparound fin missiles.**[18,19]** The information gained from the computations will hopefully aid in a successful design of the guided MLRS and future missiles equipped with wraparound fins.

Initial computations have provided interesting information about the guided MLRS missile flow field. Wind tunnel data for a similar geometry are available for comparison. The nose and canard geometries of the computational model vary slightly from the wind tunnel model. However, the results still provide insight and demonstrate the capability of ZNSFLOW for providing flow field solutions for this configuration. The computations have been run at 0° angle of attack at velocities of Mach 1.6 and Mach 2.2 and at 10° angle of attack at Mach 1.6. For all computations, each canard has a deflection of 10°. A large computational model that exceeds

24 million grid points was made for flow field computations of the missile at angle of attack. The computations demonstrated the ability of ZNSFLOW software to handle large data sets. The computational models used for the 0° angle-of-attack case exploited symmetry and were one-fourth the size of the computational model used for the angle-of-attack case.

Figure 12 shows a ZNSFLOW-computed solution of the guided MLRS missile at Mach 1.6 and 0° angle of attack. Figure 13 shows a ZNSFLOW-computed solution of the guided MLRS missile at Mach 1.6 and 10° angle of attack. The flow field changes substantially with the increased angle of attack. Figures 12a and 13a show pressure contours on a plane 1.4 calibers from the nose. This plane is just aft of the canards. The location of the vortices generated by the canard tips can be seen as small, circular low pressure regions near the canard tips. Figures 12b and 13b are 3.7 calibers from the nose. The flow field at 0° angle of attack is symmetrical, but the flow field for the 10° angle of attack is asymmetrical. Most noticeable is a large low-pressure region on the visible side of the body. Since the missile is flying at angle of attack, the deflected canard beneath the body directs more air to the visible side of the body. This low pressure region extends to the rear of the missile and is visible in Figure 13c, which is approximately 14 calibers from the nose and is just in front of the tail fins. Figure 12c is at the same location as Figure 13c. Figure 12c again shows the symmetrical flow field at 0° angle of attack in contrast to the asymmetrical flow field generated at 10° angle of attack shown in Figure 13c. In Figure 12c, the light contour shade between the dark contours near the body indicates the locations of the tail fins. The dark pressure contours in Figure 12c indicate that the position of the canard tip vortices is actually between the wraparound tail fins.



12a. Normalized pressure contours 1.4 calibers from nose tip.

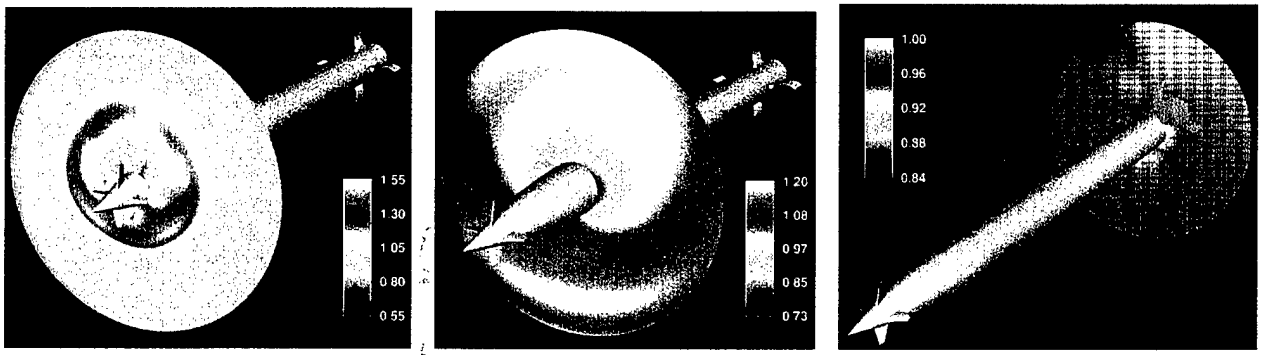
12b. Normalized pressure contours 3.7 calibers from nose tip.

12c. Normalized pressure contours 14.4 calibers from nose tip.

Figure 12. Normalized Pressure Contours at Mach 1.6 and 0° Angle of Attack.

Visualization using particle traces has also provided some insight to the guided MLRS flow field. Figure 14 shows particle traces released from the wakes of the deflected canards. The particle traces for Figure 13a were generated from a Mach 1.6 flow field solution, while the particle traces for Figure 14b were generated from a Mach 2.2 solution. Figures 14a and 14b indicate that the flow fields are similar at Mach 1.6 and Mach 2.2 at 0° angle of attack. An interesting note is that particles released at the base of the canards nearly hit the base of the tail fins. However, particles released at the canard tips are caught in a vortex that passes between the fins. The particle

traces in Figure 14c are an indication of the differences in the flow field for a guided MLRS missile at 0° and 10° angle of attack. The particle traces for Figure 14c were generated from a Mach 1.6 flow field solution at 10° angle of attack. The traces show that particles released from the canard wakes are swept to the lee side or upper side of the missile body. As mentioned earlier, the canard beneath the missile deflects more air flow to one side of the body, creating a large difference in the flow fields on the sides of the body. For the 10° angle-of-attack case, only the canard tip on the far side of the body generates a strong vortex. An indication of this vortex is the small dark circle on the left side of the body, which is visible in Figure 13b.



13a. Normalized pressure contours 1.4 calibers from nose tip.

13b. Normalized pressure contours 3.7 calibers from nose tip.

13c. Normalized pressure contours 14.4 calibers from nose tip.

Figure 13. Normalized Pressure Contours at Mach 1.6 and 10° Angle of Attack.

The ATACM-BAT multi-body problem involves the radial dispensing of several BAT sub-munitions (see Figure 15) at a low supersonic speed. This case is ideally suited for the Chimera over-set grid technique described earlier. The Chimera scheme allows each BAT to be modeled with its own simple orthogonal grid as seen in Figure 16. The trajectory of the 3-D radial dispensing sub-munitions depends on the initial ejection velocity. The flow field is complex and involves 3-D shock-boundary layer interactions and ATACM-to-BAT as well as BAT-to-BAT interactions. Detailed experimental or theoretical data were not available to help evaluate the sub-munition dispensing phenomenon for the entire BAT system, and thus the numerical solution of this problem was initiated.**[20-22]** The Chimera solution procedure was successfully used to help determine the aerodynamic interference effects.**[21]**

For a set of wind tunnel experiments, the position of the sub-munitions was set in order to evaluate flow field correction factors for nonsymmetrical dispensation at a distance near and far from the bay. The flow field correction factors are used in six-degree-of-freedom simulations of BAT dispensation for differing conditions. CFD computations were made for two configurations: Configuration A, which places the sub-munitions relatively close to the missile bay, and Configuration B, which places them farther away from the turbulence generated by the missile bay. For both Configurations A and B, there is equi-distant circumferential spacing for each sub-munition except one, which has a 5° offset. The sub-munition with the circumferential offset is

located at approximately the 11 o'clock position. Figure 17 provides a visual reference for the sub-munition positions for Configurations A and B.

Figure 14a. Particle traces for Mach 1.6, 0° angle of attack.

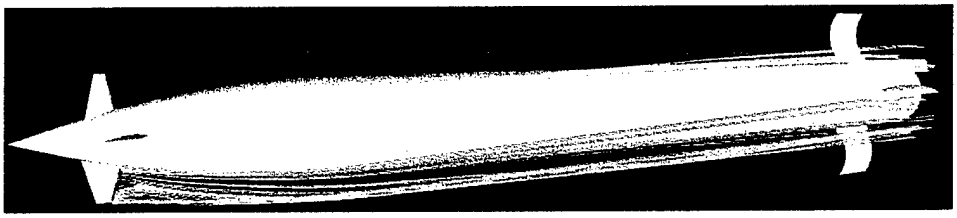


Figure 14b. Particle traces for Mach 2.2, 0° angle of attack.

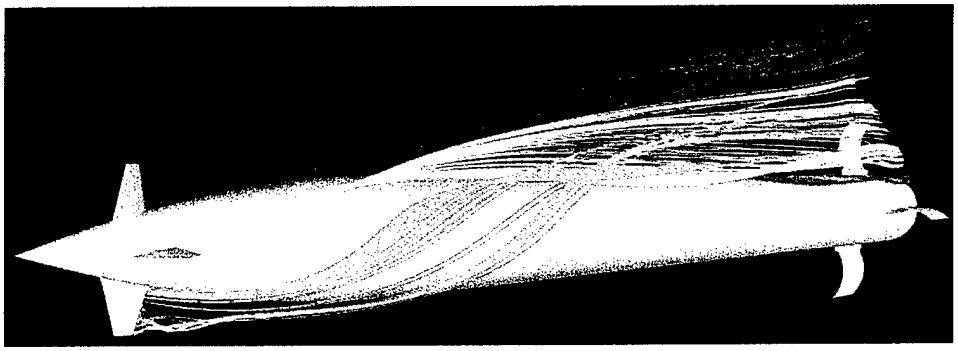


Figure 14c. Particle traces for Mach 1.6, 10° angle of attack.

Figure 14. Particle Traces at Various Mach Numbers and Angles of Attack.

4.2 Computations for BAT Sub-Munitions Ejecting From ATACM

Surface pressure contours are shown for Configuration A in Figure 18 and for Configuration B in Figure 19. The surface pressures on the Configuration A sub-munitions reveal much stronger pressure gradients than the sub-munitions in Configuration B. Also, surface pressure contours within the ATACM missile bay are somewhat different between Configurations A and B. The stronger pressure gradients on the Configuration A sub-munitions, which are much closer to the

ATACM missile bay, are indicative of the higher pitching moments generated, which tend to push the nose of the BAT sub-munitions radially inward toward the ATACM missile bay. Since the computations include multiple BAT sub-munitions, BAT-to-BAT interactions are included. These interactions are critical and have a strong effect on the aerodynamic forces and moments. The normal force and pitching moment coefficients vary between the sub-munitions, indicating the asymmetrical nature of the interacting flow field.

Figure 15. Diagram of the Multi-body System.

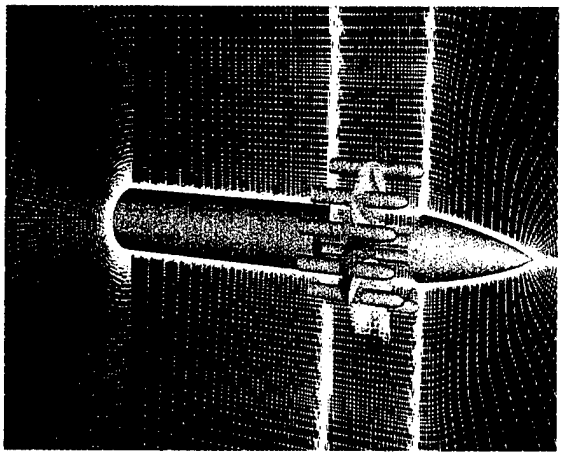


Figure 16. Grids for the BAT Sub-Munition Dispensing from ATACM.

Some experimental data **[23]** were available for comparison with the computational results of Configuration A. Figure 20 provides a visual reference for location of the BATs that were the source of the experimental data. A BAT at approximately the 5 o'clock position was equipped to record pressure data. Pressure data were collected on the side of the BAT closest to the ATACM and on the side facing away from the ATACM. On either side of the BAT, pressure data were taken at five positions. Unfortunately, the pressure data obtained from the experiment on the side of the BAT facing the ATACM do not appear to be accurate. However, the pressure coefficient data computed from the CFD solution on the side of the BAT facing the ATACM are plotted in Figure 21. Figure 22 shows a comparison between the pressure coefficient obtained from experimental and CFD-calculated data on the side of the BAT facing away from the ATACM. Both Figures 21 and 22 show the pressure coefficient as a function of the length of the BAT body in which X/L = 0 corresponds to the BAT nose and X/L = 1 corresponds to the end of the BAT body. Figure 22 shows that the pressure coefficient computed from the CFD solution is in very good agreement with experimental data. The CFD-computed data plotted in Figures 21 and 22 provide an interesting comparison that demonstrates the asymmetry of the flow field about the BAT and the strong influence of the ATACM proximity to the BAT. Although the comparison between the experimentally obtained and CFD-computed pressure coefficient is quite good, the comparisons between experimentally obtained and CFD-computed force and moments indicate that some flow field characteristics may not be captured accurately by the CFD solution.

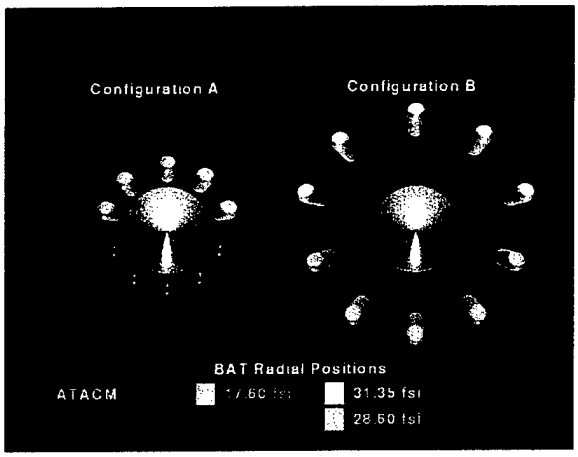


Figure 17. Configuration A and B Sub-Munition Location.

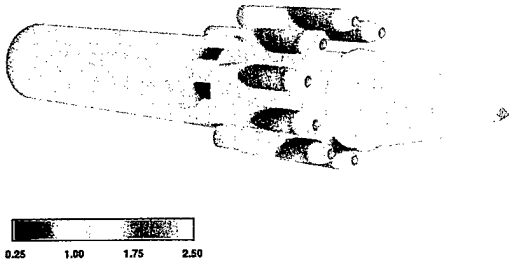


Figure 18. Normalized Surface Pressure Contours for Configuration A.

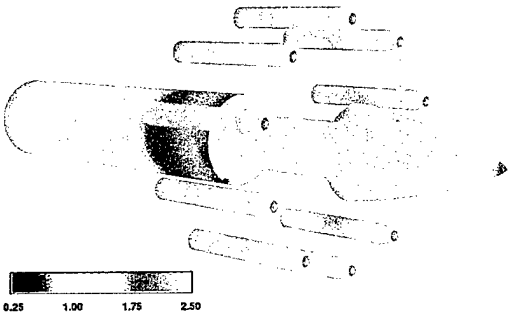


Figure 19. Normalized Surface Pressure Contours for Configuration B.

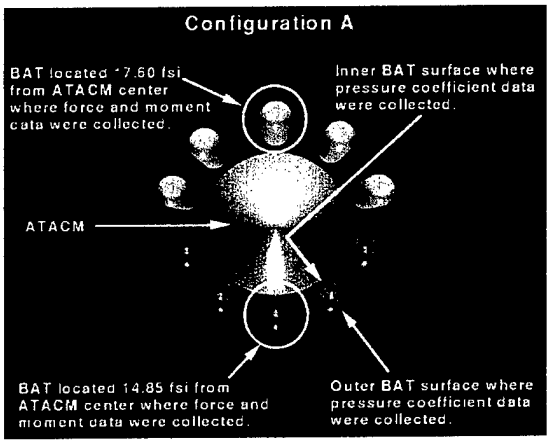


Figure 20. Locations Where Experimental Data Were Collected.

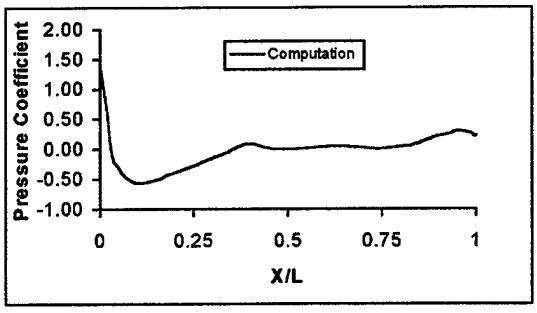


Figure 21. Pressure Coefficient Versus BAT Length for BAT Surface Facing ATACM.

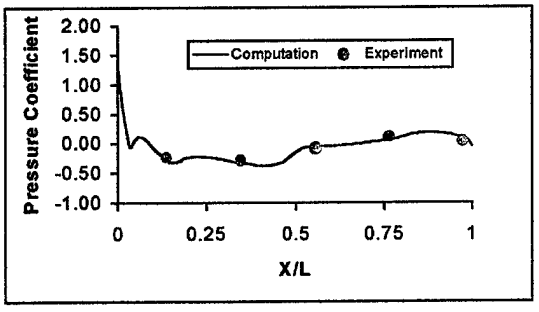


Figure 22. Pressure Coefficient Versus BAT Length for BAT Surface Facing Away From ATACM.[23]

Force and moment data were collected from the BATs located at the 12 o'clock and 6 o'clock positions. The BAT at the 12 o'clock position has a radial distance from the ATACM center of 17.60 full scale inches (fsi). The BAT at the 6 o'clock position has a radial distance from the ATACM center of 14.85 fsi. Figure 23 shows both the experimental data and the data computed from the CFD flow field solution. The data in Figure 23 indicate that the CFD-computed data match the experimental data of the BAT 17.60 fsi from the ATACM center more closely than the experimental and computed data of the BAT 14.85 fsi from the ATACM center. The data for the normal force (CN) are in good agreement for the BAT 17.60 fsi from the ATACM center. The side force (CY) data appear to be the same for the CFD-computed side force and the experimental side force. This is somewhat misleading because the magnitude of the side force is much smaller than the normal force and pitching moment, Cmz (coefficient of moment about the Z axis). The relatively small side force is a good indication that the BATs are not likely to move closer together when being ejected from the ATACM bay at 0° angle of attack. The difference between the pitching moment for experimental data and CFD-computed data is less for the BAT farthest from the ATACM. This seems to indicate increased difficulty in computing the flow field for the ATACM-BAT multi-body problem accurately when the BATs are almost in the ATACM bay.

The drag coefficient computed from the CFD solutions compares very well with the measured drag coefficient. Figure 24 shows a plot for the drag coefficient of the same BATs that were instrumented to obtain the force and moment data displayed in Figure 23. In the experiment, each BAT was mounted on a "sting." The stings were not modeled in the CFD computation. The total drag coefficient was obtained from a force measurement of the BAT with sting. The experimental value of the BAT forebody drag was estimated by taking a pressure measurement near the BAT base and using it to estimate the base drag component of the total drag coefficient. The base drag component was then subtracted from the total drag to obtain the forebody drag. An interesting note is the increase in drag with the increased distance of the BAT from the ATACM center.

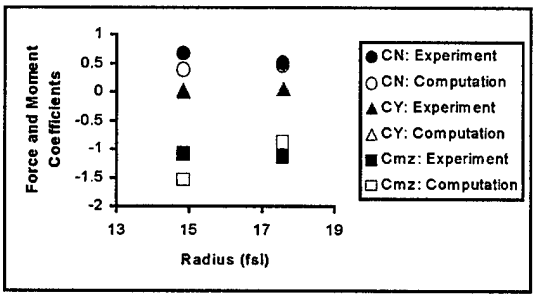


Figure 23. Force and Moment Coefficients for Configuration A.[23]

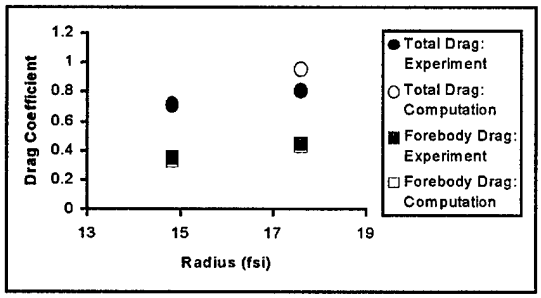


Figure 24. Drag Coefficients for Configuration A.[23]

5. ZNSFLOW USER CASES

It is also important to demonstrate that other organizations are capable of using the ZNSFLOW software. Examples of cases chosen by potential ZNSFLOW users for code demonstration are shown in this section. The two demonstration cases depicted are the theater high altitude area defense (THAAD) missile and Sea Sparrow missile. They are shown in Figures 25 and 26, respectively. The THAAD computational model was supplied by Rex Chamberlain of Tetra Research in Huntsville, Alabama.**[24]** Bob Burman of the Naval Air Warfare Center in China Lake, California, provided the Sea Sparrow computational model.**[25]** Solutions were obtained for both cases on a multi-processor SGI Onyx computer. The computational model for the THAAD missile was a three-zone, one-to-one grid point overlap computational mesh, and the Sea Sparrow computational model employed the Chimera scheme to provide communication among its five zones. Both computational models used pitch-plane symmetry. The THAAD missile computational model was built using 1,883,805 grid points, while the Sea Sparrow missile used 2,233,500 points. Each case could be run in parallel under DICE and could be visualized interactively on the Onyx computer.

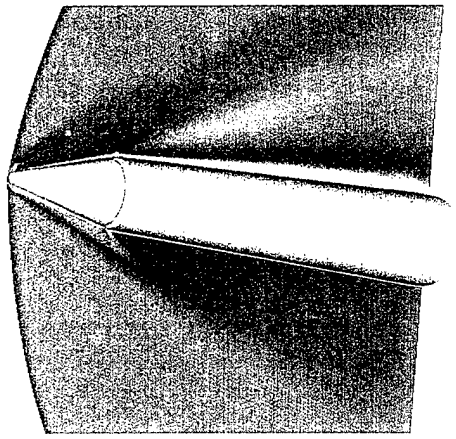


Figure 25. Mach Contours of THAAD Missile Flow Field at 10° Angle of Attack

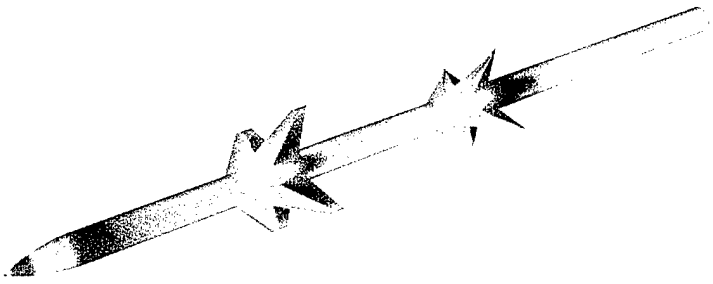


Figure 26. Surface Pressure Contours on Sea Sparrow Missile.

6. CONCLUDING REMARKS

A broad overview of the software developed under the CHSSI CFD-6 project has been presented. The scalable Navier-Stokes solver executed through the interactive computing environment, DICE, provides engineers with a fast and comprehensive CFD computation and analysis tool for complex configurations that require large computational resources. However, the solver can perform computations for simple cases just as well. The software allows the user to perform, monitor, and visualize the computations on large HPCs without copying the computational mesh and solution to their local workstation. The comprehensive interface provides control for every aspect of the computation. It was also demonstrated that the ZNSFLOW software provides accurate and visually informative results for large complex

configurations such as the guided MLRS missile and BAT dispersal from ATACM. The predictive numerical capability documented allows for accurate computation of flow fields that capture complex aerodynamic phenomena, such as interference effects, required for the improved design and modification of current and future DoD projects.

INTENTIONALLY LEFT BLANK

REFERENCES

1. Pulliam, T.H., and J.L. Steger, "On Implicit Finite-Difference Simulations of Three-Dimensional Flow," AIAA Journal, Vol. 18, No. 2, pp. 159–167, February 1982.

2. Steger, J.L., S.X. Ying, and L.B. Schiff, "A Partially Flux-Split Algorithm for Numerical Simulation of Compressible Inviscid and Viscous Flows," Proceedings of the Workshop on CFD, Institute of Nonlinear Sciences, University of California, Davis, CA, 1986.

3. Steger, J.L., F.C. Dougherty, and J.A. Benek, "A Chimera Grid Scheme," Advances in Grid Generation, edited by K. N. Ghia and U. Ghia, ASME FED-5, June 1983.

4. Benek, J.A., T.L. Donegan, and N.E. Suhs, "Extended Chimera Grid Embedding Scheme With Application to Viscous Flows," AIAA Paper No. 87-1126-CP, 1987.

5. Meakin, R.L., "Computations of the Unsteady Flow About a Generic Wing/Pylon/Finned-Store Configuration," AIAA 92-4568-CP, August 1992.

6. Goldberg, U.C., O. Peroomian, S. Chakravarthy, "A Wall-Distance-Free $k-\varepsilon$ Model With Enhanced Near-Wall Treatment," ASME Journal of Fluids Engineering, Vol. 120, pp. 457-462, 1998.

7. Baldwin, B.L., H. Lomax, "Thin Layer Approximation and Algebraic model for Separated Turbulent Flows," AIAA 78-257, January 1978.

8. Ferry, E.N., J. Sahu, and K.R. Heavey, "Navier-Stokes Computations of Sabot Discard using Chimera Scheme," Proceedings of the 16th International Symposium on Ballistics, September 1996.

9. Sahu, J., K.R. Heavey, and E.N. Ferry, "Computational Fluid Dynamics for Multiple Projectile Configurations," Proceedings of the 3rd Overset Composite Grid and Solution Technology Symposium, Los Alamos, NM, October 1996.

10. Sahu, J., K.R. Heavey, and C.J. Nietubicz, "Time-Dependent Navier-Stokes Computations for Sub-munitions in Relative Motion," Proceedings of the 6th International Symposium on Computational Fluid Dynamics, Lake Tahoe, NV, September 1995.

11. Almasi, G.S. and A. Gotlieb, "Highly Parallel Computing," Second Edition, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

12. Pressel, D. M., "Results from the Porting of the Computational Fluid Dynamics Code F3D to the Convex Exemplar (SPP-1000 and SPP-1600)," ARL-TR-1923, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, March 1999.

13. Birch, T., private communication, DERA, Bedford, UK, 1995.

14. Sturek, W., T. Birch, M. Lauzon, C. Housh, J. Manter, E. Josyula, and B. Soni, "The Application of CFD to the Prediction of Missile Body Vortices," AIAA 97-0637, January 1997.

15. Weinacht, P., "Documentation and User's Guide for the ZNSFLOW Code," to be published but available upon request.

16. Clarke, J., C.E. Schmitt, J.J. Hare, "Developing a Full Featured Application from an Existing Code Using the Distributed Interactive Computing Environment," Proceedings of 1998 DoD HPC User's Group Conference, June 1998.

17. Clarke, J., "Network Distributed Global Memory for Transparent Message Passing on Distributed Networks," ARL-CR-173, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, 1994.

18. Edge, H., "Computation of the Roll Moment Coefficient for a Projectile With Wraparound Fins," ARL-TR-23, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, 1992.

19. Patel, N., H. Edge, J. Clarke, "Three-Dimensional (3-D) large Fluid Flow Computations for U.S. Army Applications on KSR-1, CM-200, CM-5, and Cray C-90," ARL-TR-712, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, 1995.

20. Wooden, P. A., W. B. Brooks, J. Sahu, "Calibrating CFD Predictions For Use In Multiple Store Separation Analysis," AIAA Paper No. 98-0754, January 1998.

21. Sahu, J., H.L. Edge, K.R. Heavey, E. Ferry, "Computational Fluid Dynamics Modeling of Multibody Missile Aerodynamic Interference," Proceedings of the NATO RTO AVT Symposium on Missile Aerodynamics, Sorrento Italy, May 1998.

22. Wooden, P.A., E.R. McQuillen, W. B. Brooks, "Evaluation of a Simplified Multiple Store Interference Model," AIAA Paper No. 98-2800, June 1998.

23. Lee, P.J., "Analysis Report of Army TACMS Block II Captive Airloads Wind Tunnel Data from HSWT Test 1218," 3-18400/6R-050, Lockheed Martin Vought Systems, Dallas, TX, November 1996.

24. Chamberlain, R., Private communication, U.S. Army Research Laboratory, MD, 1999.

25. Burman, B., Private communication, U.S. Army Research Laboratory, MD, 1999.

26. Goldberg, U., and D. Apsley, "A Wall-Distance-Free Low Re $k-\varepsilon$ Turbulence Model," Computer Method and Applied Mechanical Engineering, pp. 145-227, 1997.

27. Bailey, David H., "RISC Microprocessors and Scientific Computing," Proceedings for SUPERCOMPUTING 93, Association for Computing Machinery, Portland, Oregon, 1993.

28. Speech of Dr. J. Boris at the DOD HPC MOD program Annual Users Group Meeting held at NRL in 1996 summarizing the state of the art.

APPENDIX A

TURBULENCE MODELS USED IN ZNSFLOW

INTENTIONALLY LEFT BLANK

TURBULENCE MODELS USED IN ZNSFLOW

ZNSFLOW has three options for modeling turbulence: the Baldwin-Lomax model, the one-equation point-wise turbulence model, and two-equation point-wise turbulence model. Following is a brief description of the models' formulation.

1. BALDWIN-LOMAX MODEL

This is an algebraic, two-layer model with the attractive feature of removing the necessity of determining the displacement thickness or the wake thickness and instead uses the distribution of vorticity to determine the length scales in the outer model.

The model is subdivided into an inner and an outer model. The inner model is applied between the body surface and a cross-over point where the inner viscosity exceeds the viscosity evaluated using the outer model. The outer model is applied outward from the cross-over point. The inner model employs the Van-Driest mixing length approach and uses the following:

$$(\mu_T)_{inner} = \rho\, l^2 \,|\, \omega \,| \tag{1}$$

in which

$$l = ky\left[1 - \exp(-y^+ / A^+)\right] \tag{2}$$

Here, y is the coordinate normal to the surface and $|\ \omega\ |$ is the magnitude of the local vorticity. The constants, k and A^+, have the values 0.4 and 26.0, respectively. The nondimensional boundary layer coordinate, y^+, is a function of the fluid viscosity ν_w, fluid density ρ_w, shear stress τ_w, and the dimensional distance from the wall, y. The subscript w indicates that the quantities are to be evaluated at the body surface. For wake flows, the exponential term shown above is set to zero.

$$y^+ = \frac{y u_*}{\nu_w} \tag{3}$$

$$u_* = \sqrt{\tau_w / \rho_w} \tag{4}$$

In the outer region, the model takes the form

$$(\mu_T)_{outer} = \rho K C_{cp} F_{wake} F_{KLEB}(y)$$

in which F_{KLEB} (y) is the Klebanoff intermittency factor which can be written as

$$F_{KLEB}(\mathrm{y}) = \left[1 + 5.5\left(\frac{C_{KLEB}}{y_{\max}}\right)^6\right]^{-1} \tag{5}$$

and $K = 0.0168, C_{cp} = 1.6, C_{KLEB} = 0.3$.

The parameter F_{WAKE} is evaluated as

$$F_{WAKE} = smaller\ of\ \left[(y_{\max}F_{\max})\ldots or\ldots\left(C_{WK}\,y_{\max}u_{DIF}^2 / F_{\max}\right)\right] \tag{6}$$

in which u_{DIF} is the total velocity difference across the boundary layer or wake and C_{WK} was originally assigned a value of 0.25 by Baldwin and Lomax, although more recently, a value of 1.0 has been used. $F_{\max}$ is determined from the maximum value of the function $F(y)$, evaluated from

$$F(y) = y|\omega|\left[1-\exp(-y^+ / A^+)\right]. \tag{7}$$

While $y_{\max}$ is the value of y whereby $F(y)$ equals $F_{\max}$. For wake flows and separated boundary layers, attention needs to be paid to the appropriate normal direction. In attached boundary layer calculations, $F_{WAKE} = y_{\max}F_{\max}$ is used.

2. POINT-WISE TURBULENCE MODEL

To overcome the ambiguity of the wall distance in turbulence formulations, Goldberg et al. **[26]** proposed the use of wall proximity indicators that are local, i.e., point-wise in nature and that indicate the influence of the walls indirectly through parameters. These wall-distance-free models are tensorially invariant and frame indifferent, making them applicable to arbitrary topologies and moving boundaries. They have been shown to be independent of structured and unstructured meshes and computer architecture, including massively parallel machines.

The following is based on and summarizes Goldberg's "Summary of Linear Topology-Free One- and Two-Equation Turbulence Models."

2.1 Formulation of Wall-Distance-Free Turbulence Models

2.1.1 *One-Equation Model*

The one-equation model consists of solving the transport equation for the undamped eddy viscosity (R):

$$\rho\frac{DR}{Dt} = \frac{\partial}{\partial x_j}\left[\left(\mu + \frac{\mu_t}{\sigma_R}\right)\frac{\partial R}{\partial x_j}\right] + C_1\rho(RP_k)^{1/2} - (C_3 f_3 - C_2)\rho D \tag{8}$$

in which P_k is the turbulence production expressed in terms of the Boussinesq model and D is the destruction term. The eddy viscosity field is given by

$$\mu_t = f_\mu \rho R \tag{9}$$

in which

$$f_\mu = \frac{\tanh(\alpha\chi^2)}{\tanh(\beta\chi^2)} \tag{10}$$

$$\chi \equiv \frac{\rho R}{\mu} \tag{11}$$

The damping function

$$f_3 = 1 + \frac{2\alpha}{3\beta C_3 \chi} \tag{12}$$

is derived from asymptotic arguments, and the value of $C_3 = 1.146$, with the value of $\beta = 0.2$, and $\alpha = 0.07$. Equation (8) is subject to the boundary condition that on solid walls $R = 0$, while free stream inflow and initial conditions demand $R_\infty \leq \nu_\infty$.

2.1.2 *Two-Equation Model*

The Reynolds stresses are related to the mean strain gradients through the Boussinesq model, and the eddy viscosity contains a damping function. For details of the derivation, the reader is referred to in reference **[6]**. The turbulence kinetic energy and the dissipation rate, k and ε, are determined by the transport equations

$$\frac{\partial(\rho k)}{\partial t} + \frac{\partial}{\partial x_j}(U_j \rho k) = \frac{\partial}{\partial x_j}\left[\left(\mu + \frac{\mu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_j}\right] + P_k - \rho\varepsilon \tag{13}$$

$$\frac{\partial(\rho\varepsilon)}{\partial t} + \frac{\partial}{\partial x_j}(U_j \rho\varepsilon) = \frac{\partial}{\partial x_j}\left[(\mu + \frac{\mu_t}{\sigma_\varepsilon})\frac{\partial \varepsilon}{\partial x_j}\right] + (C_{\varepsilon 1} P_k - C_{\varepsilon 2}\rho\varepsilon + E)T_t^{-1} \tag{14}$$

in which P_k is the turbulence production modeled following the Boussinesq concept. The realizable time scale is

$$T_t = \frac{k}{\varepsilon}\max\{1, \frac{C_\tau}{\sqrt{R_t}}\} \tag{15}$$

The model includes an extra source term, E, that is designed to increase the level of ε in non-equilibrium flow regions. This reduces the length scale and improves the prediction of adverse pressure gradient flows.

$$E = A_E \rho \nu \sqrt{\varepsilon T_t}\,\Psi \tag{16}$$

in which

$$\Psi = \max\{\frac{\partial k}{\partial x_j}\frac{\partial \tau}{\partial x_j}, 0\} \tag{17}$$

and

$$v = \max\{k^{1/2}, (v\varepsilon)^{1/4}\} \tag{18}$$

The model constants are $C_\tau = \sqrt{2}$ and $A_E = 0.3$. The boundary conditions are as follow: at walls, the kinetic energy of turbulence and its first normal-to-wall derivative vanish. The boundary condition for ε is based on its near-wall asymptotic behavior, i.e.,

$$\varepsilon_w = 2v_1 \frac{k_1}{y_1^2} \tag{19}$$

in which "1" denotes the first internal node. This boundary condition implies that $(\partial k / \partial y)_w = 0$, thus satisfying the second boundary condition for k implicitly.

APPENDIX B

COMPUTER SCIENCE ISSUES BEHIND THE SUCCESS OF CHSSI PROJECT CFD-6

INTENTIONALLY LEFT BLANK

COMPUTER SCIENCE ISSUES BEHIND THE SUCCESS OF CHSSI PROJECT CFD-6

1. INTRODUCTION

When the CHSSI PROJECT CFD-6 was first conceived, two important software issues and one important hardware issue had yet to be resolved, which would have a strong bearing on the success or failure of this project:

a. What (if any) benefit do large memory high performance computing jobs receive from the benefit of a memory hierarchy involving one or more levels of cache memory?

b. Can implicit CFD codes be successfully parallelized without damaging their convergence properties and/or requiring significant modifications of the algorithm?

c. Assuming that large memory high performance computing jobs do benefit from the presence of cache memory, which is better: large caches or faster access to the main memory?

2. MEMORY SYSTEM ISSUES AND SERIAL EFFICIENCY

According to David Bailey (formerly at NASA Ames Research Center), his experiments on Intel i860 processors showed that caches were of limited or no value to large memory high performance computing jobs.**[27]** Based on this work and those of several other major researchers in the fields of computer architecture and/or the computational sciences, the benefit of cache memory was highly dubious. This was a disturbing conclusion since this project was expecting to use scalable parallel processors based on RISC processors and dynamic random access memory (DRAM). The problem with this is that the speed of the processors was increasing rapidly, while the bandwidth to memory was increasing slowly, and the memory latency was barely changing at all. Therefore, unless cache memory could be shown to be of value, it was expected that it would be only a matter of time before continued improvements in the peak speeds of the processors would become irrelevant.

Early experiments in running code (not optimized for the SGI RISC processors) were not very encouraging. These experiments were run using a single processor of a 75-MHz (300 Mflops[4] peak speed) R8000-based SGI Power Challenge. Compared to the performance of running an in core version of the F3D code on one processor of a Cray C-90, it was hoped that the code running on the Power Challenge would decelerate by roughly a factor of 3. In fact, a deceleration of roughly a factor of 45 was observed. Further experiments based solely on the use of compiler options (run by Daniel Pressel) proved to be of limited benefit. Clearly at this time, things were not looking good for architectures based on DRAM main memory and one or more levels of cache.

[4]One million floating point operations per second

At that time, efforts had been aimed at minimizing the changes in the code in the hopes of having a single program that would perform well on both vector processors and scalable parallel processors. This approach was abandoned, and traditional tools (e.g., profiling) were applied to the problem. The concept here was that vectorizable code was designed to run on vector processors (Cray vector processors, to be specific). As a result, the code was written in a manner that would keep the vector processor busy. At the same time, issues such as locality of reference or required memory bandwidth were of little concern since these machines made few assumptions concerning locality of reference, supported very high levels of memory bandwidth, and vector processing was inherently tolerant of moderate levels of memory latency.

Using these tools, it was easy to see where the hot spots were in the program. Further analysis produced the following conclusions:

a. Many of the loops in the program were accessing the large arrays with a large stride. Frequently, this resulted in a high cache miss rate. In all cases, it resulted in a high TLB miss rate, which was just as bad (TLB is the part of the memory system that maps addresses from the logical address space of a program to the physical address space of main memory). This problem was addressed through a combination of four main techniques:

(1) Reordering the indices of the arrays;

(2) Reordering the loops in the loop nest;

(3) The judicious use of matrix transpose operations (for invariant/relatively invariant arrays, the matrix transpositions should be kept in memory in between uses and revised only when they need to be); and

(4) Blocking.

b. Some cases of copying between large arrays were relics of the code since it was written as an out-of-core solver. In general, it was possible to eliminate this copying. While originally this was not productive, had this not been done, it could have easily represented half of the remaining run time when the other optimizations were performed.

c. Some of the loops were using scratch arrays to pass data from one loop to the next. The main justification for not merging the loops was that the merged loop was too complicated for the vectorizing compiler to automatically vectorize. Since this was no longer a concern, the need to use these arrays represented a performance problem; the loops were merged, and the scratch arrays were eliminated.

d. When cache-based architectures are used, it is highly desirable to perform as many calculations on a set of values as possible, before moving to the next set of values. This is in sharp contrast to vector codes wherein one wants to maximize the number of times the same operation/set of operations can be performed on independent sets of values. As a result, vector-oriented code will inherently require a much higher memory bandwidth to obtain the same level of performance. Two related examples of how this information can be used are

(1) If one has a loop nest such as:

```
DO ... M=1,5
 DO ... N=1,5
  DO ... L=1,LMAX
   DO ... K=1,KMAX
    Several lines of code involving arrays such as A(K,L,N,M).
```

It will probably produce far fewer cache and TLB misses if the loop nest can be rewritten as

```
DO ... L=1,LMAX
 DO ... K=1,KMAX
  DO ... M=1,5
   DO ... N=1,5
    Several lines of code that now involve arrays such as A(N,M,K,L).
```

This will also have the added benefit of potentially supporting more aggressive forms of loop unrolling.

(2) If one has a long complicated loop that employs all the values associated with a single data point, then it is more efficient to store those values in an array Q(N,J,K,L) than in an array Q(J,K,L,N) in which N is some small integer value such as 5 or 6. Similarly, if one has two or more arrays that are always used as a group, then those arrays should be merged. An example of this is merging the arrays XX(J,K,L), XY(J,K,L), and XZ(J,K,L) into a single array XXYZ(3,J,K,L).

e. When vector-based architectures are used, it is sometimes necessary to process a plane of data at a time in order to avoid limitations caused by dependencies. This can result in the use of scratch arrays that are too big to fit in cache. However, if one accepts that the code no longer needs to be vectorizable, then one can process just a single row or column of data at a time. This will normally shrink the size of the scratch arrays by one or more orders of magnitude. At this point, the array has the advantage of being "locked" into cache. The one catch is that the cache still needs to be big enough. Many processors have relatively small primary caches ranging in size from 8 KB to 64 KB. In some cases, the scratch arrays might not fit into the primary cache, or at best would be a tight fit, which would result in an undesirable level of cache thrashing. However, if the processor has a large off-chip cache (e.g., at least 1 MB in size), then it is possible to "lock" the scratch arrays into cache with plenty of room to spare. Note that this observation can also be important when blocking code, since very small block sizes may be of little or no value.

f. It was also found that a small number of the loops were expensive and computationally intensive but had a low cache miss rate. For these loops, highly aggressive techniques were used to improve the level of register reuse and to take better advantage, in other ways, of the pipelined nature of RISC processors.

g. Three final observations about this process were

(1) Tuning is an iterative process.

(2) For programs that are not written in an object-oriented fashion (e.g., Java or C++), relatively few subroutines or functions will need to be tuned. However, if the tuning process is successful, the number of routines requiring tuning will increase (e.g., by a factor of 2 or 3, or from 5-10 to 20-30 routines) as the process finishes.

(3) The 80-20 rule (20% of the work gets 80% of the benefit) does not apply. Many of these systems have very significant increases in performance as one passes the 90% to 95% tuned point. The reason for this has to do with the high cost of cache and TLB misses (100 or more cycles). The point here is that until one has reduced the cache miss rate (in terms of the misses that miss all the way back to main memory) to 1% or less, most other forms of tuning will not provide a significant gain in performance. However, if the overall cache miss rate is tuned to this extent, then other forms of tuning may be worth considering (at least for the two or three most expensive routines).

The net result of these manipulations was to accelerate the code by a factor of 11 while leaving the algorithm and the convergence properties unchanged. It is also important to note that all of this was done before attempting to parallelize the code, so this acceleration was not achieved by using substantially more hardware. Additionally, since independent measurements on the Cray C-90 showed that the original vector-optimized code achieved 30% to 40% of peak, these improvements in performance were not the result of starting from a poorly written/optimized code.

At this point, we have answered the first concern. It is actually possible for a large memory high performance computing job (we have run jobs as large as 73 GB) to benefit from a cache, providing that the cache is at least 1 MB in size (2- to 8-MB caches may be even better). The third concern had to do with trade-offs between fast memory access and the presence of cache. The ratio between memory latencies of the fastest versus the slowest systems in this market is roughly 2:3. Similarly, the ratio for memory bandwidths is roughly 2:4. Clearly, for well-tuned code, the large cache can be advantageous and is a better choice than to worry about minor improvements in memory latency and memory bandwidth. However, for untuned code, the reverse statement can be made. The problem is that even on the systems with the fastest DRAM-based memory systems, the performance of untuned/poorly tuned code is likely to leave a lot to be desired.

3. PARALLELIZATION ISSUES

When this project was started, it was "common knowledge" that implicit CFD codes (e.g., F3D) could not be efficiently parallelized without adversely affecting the convergence properties of the algorithm.**[28]** To one of the authors (Daniel Pressel), this seemed like a rather strange statement. It was known that F3D performed efficiently on a Cray C-90, a vector processor. Since vectorization is a form of parallelism, it should theoretically be possible to exploit the same parallelism with non-vector processors in order to demonstrate parallel performance. Upon further reflection, there were three straightforward reasons for the "common knowledge":

a. Commonly used approaches to parallelization assumed that one had a virtually infinite level of available parallelism. This assumption was necessary because of the limited performance of the individual processors being used. In contrast, Cray vector processors were reasonably efficient even when the levels of available parallelism were in the range of 50 to 100.

b. Ordinary techniques for parallelizing CFD codes were based on the concept of domain decomposition. This had the benefit of generating a good level of locality of reference, which helped to reduce the amount of data motion between processors in a distributed memory environment (such as is normally used in highly scalable parallel processors). Unfortunately, when this approach to implicit CFD codes was applied in a naive manner, the convergence properties of the code were frequently adversely affected when as few as 32 processors were used.

c. The obvious alternative was to use some form of loop-level parallelism. What made this obvious is that vectorization is a form of loop-level parallelism. Unfortunately, when attempting to implement loop-level parallelism in a distributed memory environment, one is likely to have a very poor locality of reference. Furthermore, the messages between the processors are likely to be small in size, but huge in number. This will make the program perform poorly on most of the massively parallel systems on the market. The two solutions are

(1) Use a shared memory system, which would eliminate the performance problems associated with using a distributed memory system.

(2) Use a distributed memory system that supports a particularly efficient version of Software Virtual Shared Memory. The Cray T3D was designed to be just such a system.

One might question why the first of these approaches had not been tried before. The answer is quite simple. Before this project began, there were three types of shared memory systems on the market:

a. Systems based on two to eight mini-computer/main frame processors that were poorly suited for the task (e.g., too weak and/or too expensive for the delivered level of performance).

b. Vector-based systems using 2 to 16 processors. The biggest problem here was that the vector processors were already using much of the available parallelism, leaving little hope of showing significant levels of speedup.

c. Shared memory systems based on micro-processors using as many as 30 processors. Unfortunately, the performance of these processors was limited, resulting in systems that were once again poorly suited for the task.

Starting in the early 1990s, all this started to change. SGI announced the R8000 processor would have a peak speed of 300 Mflops. The Digital Equipment Corporation (DEC) produced the 21064 Alpha processor with a peak speed of 150 Mflops; IBM announced the POWER2 processor with a peak speed of 267 Mflops; and HP produced a processor rated at 200 Mflops. SGI, DEC, and Convex (using HP's processor) all produced shared memory systems of various

sizes based on these powerful processors. Additionally, Cray Research produced the Cray T3D based on the Alpha processor, which was designed to support the CRAFT programming model (a form of Software Virtual Shared Memory). In theory, the SGI and Convex systems should have been equivalent in performance to a small Cray C-90, while Cray claimed that a large T3D was more powerful than a 16-processor Cray C-90 (at least for some problems).

With the arrival of eight SGI Power Challenges (collectively referred to as the Power Challenge Array) at the U.S. Army Research Laboratory (ARL), and with the Army High Performance Computing Research Center gaining access to a large Cray T3D, the scene was set to see what these machines were capable of doing. Daniel Pressel was selected to optimize the F3D code for the Power Challenge, while Marek Behr was selected to do the same for the Cray T3D. As mentioned earlier, the only startling thing about the performance of this code on the Power Challenge was the poor level of the performance. Furthermore, these machines arrived with 12 processors each, were upgradable to a maximum of 18, and did not support the shared memory-programming paradigm when used as a cluster. Therefore, it would be impossible to overcome this performance deficit by using large numbers of processors.

Marek Behr's problems with the Cray T3D were even more serious. Whereas the Power Challenge was able to run the code using a single processor from the first (albeit very slowly), that option did not even exist for the Cray T3D. Therefore, it was impossible for him to even consider the issues surrounding serial efficiency until after he had successfully parallelized the code. Furthermore, once he had succeeded in parallelizing the code using the CRAFT model, the performance shortcomings of this model became all too apparent. By this time, much of the serial tuning for the Power Challenge had been completed and the use of compiler directive based loop-level parallelism was starting to show significant levels of speedup and overall levels of performance. At this point, rather than stopping, Marek Behr decided to take the extreme step of manually implementing loop-level parallelism on the Cray T3D using message-passing code. By using the SHMEM (Cray's so-called "shared memory" primitives, also known as single-sided message-passing primitives), he was able to demonstrate a level of performance that several times exceeded the performance when the CRAFT model was used. While the per-processor level of performance was less than had been hoped for, it was now at least high enough that it would be possible to achieve acceptable levels of performance for all but the smallest problems by using larger numbers of processors.

Daniel Pressel then tried to extend the shared memory version of the code to support the Convex Exemplar, which claimed to be a simple extension of the shared memory environment. Unfortunately, while this claim was largely correct, he was never able to achieve acceptable levels of performance when using more than eight processors (although the performance with eight processors was superior to that of a single processor of a Cray C-90). Following this, work began to test an older version of the SGI Challenge at ARL and a newly arrived R10000-based SGI Challenge at the Tank-Automotive Command (TACOM) Distributed Center. Both of these efforts were fully successful and allowed us to develop additional code modification to support a wider range of system configurations.

Following this, the shared memory version of the code was rapidly transferred to the SGI Origin 2000 when it arrived at the newly created ARL Major Shared Resource Center (MSRC). At about the same time the Army High Performance Computing Research Center (AHPCRC), which was now referred to as a distributed center, gained access to a succession of Cray T3Es. The distributed memory version of the code was rapidly transferred to these systems. After this, significant additional improvements were demonstrated on the Origins as larger and faster systems were brought on line. Currently, the largest Origins in the Department of Defense (DoD) HPC modernization program have 128 processors, with plans under way to create some 256-processor systems. Unfortunately, it is not clear to what extent this code will be able to take advantage of the 256-processor system. Also, some of the systems have been upgraded from 195-MHz (390 Mflops) processors to either 250-MHz (500 Mflops) or 300-MHz (600 Mflops) R12000 processor-based systems.

At the same time, the AHPCRC and the MSRC at the U.S. Army Engineering Research and Development Center (ERDC) obtained Cray T3E 1200s with more than 200 processors each, while the MSRC at Naval Oceanographic Office (NAVO) has taken delivery of a Cray T3E 900 with more than 900 processors in it. While for most problems this is more hardware than is reasonable to use for just one job, it has allowed Marek Behr to demonstrate highly desirable levels of performance on these machines. Subsequently, he ported this version of the code to the SGI Origin 2000 (although at a lower level of performance than that achieved with the shared memory version of the code). He also created a version of the code that uses only MPI calls for use on the IBM SP. Unfortunately, tests run with this code on the IBM SP at the ERDC MSRC have proved to be somewhat disappointing. Presumably, this is the result of the IBM SP having a larger latency when passing messages between processors, which severely impedes the performance of codes such as this one, which frequently passes huge numbers of small messages. A more detailed description of this effort is given in Section 2.3.2 in the main body of this report.

4. PERFORMANCE METRICS FOR PARALLEL PROGRAMS

Traditionally, talks concerning the performance of parallel programs have stressed the scalability of the program(s) being discussed. This project has taken a very different point of view. It has been based on the assumption that from the standpoint of performance, the two things that really matter are

a. When you are talking about a single job, what matters is the time to completion.

b. When you are talking about a series of jobs, what matters is the overall throughput. This is a function of how efficiently the hardware is being used, the performance of a single job on the hardware (note in this case, one can frequently achieve better levels of throughput by using fewer processors per job but running more jobs at once with a higher level of parallel efficiency), and the amount of hardware that is available upon which to run the jobs.

This does not mean that we do not consider parallel speedup and parallel efficiency to be important. Rather, we consider it to be only part of the whole story. The remaining parts of the story are

a. The raw performance of the processors;

b. The number of processors that are readily available (this is a function of usage by other users, system configurations, the amount of hardware that was purchased, and probably other factors as well);

c. Serial efficiency. If the job is not efficiently using its processors, then scaling to larger numbers of processors will be of questionable value. Our experience (as well as that of others) with the CRAFT model on the Cray T3D is an excellent example of this point.

d. The efficiency of the algorithm being used. Many of the early success stories of parallel computing involved algorithms that were really inefficient but were also extremely easy to parallelize (e.g., Monte Carlo methods). As a result, one could get very high levels of floating point performance and still have a slowly running job.

e. Any hidden inefficiencies associated with the parallelization of the algorithm. A simple example of this is to perform the same calculation on every processor to avoid the need for communication. This can improve the run time of the job, but it can also inflate the operation count. The program should get credit for the faster run time, but one needs to discount the added operations before calculating things like Mflops or serial efficiency. In other cases, the parallelization technique might make the operation count a function of the number of processors being used (e.g., O[log{n}]). This can still result in parallelization being a success; however, it will be a much smaller success (assuming that this occurs in a key portion of the program).

Our reasons for being concerned with these issues are simply stated; we made every effort to achieve very high levels of performance based on wall clock time, knowing full well that there would be significant limits on the scalability of these codes. As a result, the performance of the shared memory code running on the Origin 2000 exceeds that of the distributed memory version of the code running on a Cray T3E-1200 (when using the same numbers of processors), even though the Cray T3E-1200 is rated (and sold) on a per-processor basis as being three times as fast as the Origin 2000. Furthermore, when looking at other benchmarks run on various versions of the Cray T3E (all of which were supposed to have faster processors than the Origin 2000), we find that the per processor Mflops delivered by our code is superior to what other researchers are seeing for their codes on the Cray T3E. Finally, researchers at the National Center for Atmospheric Research (NCAR) have reported that one of their climate models runs noticeably faster on an Origin than it does on the Cray T3E (like our code, this model is not highly scalable, so this is an important result).

5. LIMITATIONS ON THE PARALLEL PERFORMANCE OF THE CODE

Several limitations are inherent with the use of loop-level parallelism, which will limit the achievable levels of parallel speedup when larger numbers of processors are used. Taking a quick look at these limitations,

a. Since we are writing the individual loops to execute in parallel, the available parallelism is limited by the number of iterations in the loop. In some cases, it is possible to merge two or

more loops in a loop nest to effectively eliminate this limitation (something that was done with the distributed memory version of the code but not the shared memory version of the code). However, if there is a loop dependency in all but one direction, it will not be possible to make this transformation. Furthermore, if some of the loops are prohibited from making the transformation, then making the transformation on the remaining loops may be of less value.

b. An overhead cost is associated with getting in and out of parallel sections of code. At the same time, loop-level parallelism will almost always have significantly less work per synchronization/communication event than is observed in message passing code based on domain decomposition. As a result, on a shared memory system it may be desirable to leave some of the loops unparallelized (this may not be an option in a distributed memory environment). This is especially likely to be the case in some of the boundary condition routines. Unfortunately, when 100 or more processors are used, this serial code is likely to dominate the run time (Amdahl's Law).

c. There can also be performance issues when one is writing code loops with moderate amounts of work when the number of processors approaches the available level of parallelism. In general, the overhead cost need not dominate the performance of these loops; however, it may have an impact that cannot be entirely ignored. This effect is comparable to the way the ratio between computation and communication becomes unfavorable for traditional codes as the number of processors increases (for fixed size problems).

d. A direct result of parallelizing some or all of the loops in only one direction is that the available parallelism is roughly equal to the cube root of the number of grid points in the zone being processed (square root for 2-D problems). This violates one of the key assumptions of scaled speedup, that the available parallelism is proportional to the problem size. As a result, any metrics based on the concept of scaled speedup are of limited applicability.

e. Another direct consequence of limited amounts of parallelism is that a plot of performance (or if one prefers, speedup) as a function of the number of processors being used will have a staircase appearance for large numbers of processors. For example, if a loop has 100 units of parallelism, then jobs using 25 to 33 processors will all run at the same speed. Jobs using 34 to 49 should run 33% faster than that. Jobs using 50 to 99 will run twice as fast as the 25-processor job. Finally, when 100 processors are used, the peak speed of 4 times as fast as the 25-processor job will be obtained. It is important to understand that this effect does not depend on hardware limitations, nor is it an example of Amdahl's Law. It is an inherent result of integer division. Of course, the extent to which this effect is actually observed is likely to be complicated by other factors that are discussed in Section 6 of this appendix, so the predicted performance increases are really for the ideal case.

6. ADDITIONAL CONSIDERATIONS

The previous list of limitations is entirely theoretical in nature. In addition to them, there are the following practical considerations:

a. Many jobs will perform I/O at the beginning and/or the end of every run. In general, this I/O will not be parallelized (some systems still do not support parallel I/O, while on many systems, it can be counter productive).

b. It takes time to allocate memory (especially for large memory jobs using gigabytes of memory). Furthermore, on some systems, the run time for a job will include the time required for the system to de-allocate the memory after the job has completed.

c. It is common practice when performing timed runs for benchmarking purposes to keep the runs short (as opposed to running them to convergence).

If one normally executes large numbers of time steps in a single run (e.g., going to convergence in just one run), then it might be totally reasonable to include the initialization and termination costs when performing benchmark runs. This can also be the case when one is trying to benchmark the performance of one or more jobs on a system for procurement purposes. However, for the purpose of benchmarking CHSSI software, we feel that it is best if these effects have been subtracted from the run time. While there are many ways in which this can be done, the simplest one, and the one that has been used for this report is to do two sets of runs. One involving larger numbers of time steps (e.g., 50 to 100) and one involving smaller numbers of time steps (e.g., 1 to 10). On a suitably quiet system, subtracting one set of runs from the other should give a good estimate of the asymptotic behavior of the code per time step. On a system shared with other users, the results are likely to have some noise, although the amount of noise need not be large if the system is not overloaded (on overloaded systems, a number of effects come into play that will make it all but impossible to get useful numbers).

7. THE QUESTION OF METRICS

As with any good research and development project, one needs metrics for judging the success of a project. One approach would be to compare the wall clock time of this code to that of other codes doing the same problem. This would have the strong advantage of helping to test the claims that we had an efficient serial implementation and an efficient algorithm, and therefore might not need the same level of parallelism to be of significant value. For whatever reason, this has not been done.

A second possibility, and one that was heavily used at the start of this project, is to compare the performance (again in terms of wall clock time) for the various combinations of system/ versions of the code against each other and against the performance of the original vector-optimized code on both the C-90 (where it was exceptional) and on the SGI Power Challenge (where it performed poorly). Again, if one assumes that the performance of a single processor of a C-90 provided acceptable levels of performance for a particular problem, then this approach can allow one to estimate which problems can be tackled on each system and how much hardware will be required. This approach also has the advantage of allowing one to look at real-world considerations such as the availability of sufficient memory to run the job on the different

platforms, and the time spent in the queue. While this approach has significant merit, its importance has been substantially diminished.

A third possibility is to measure scaled speedup (this is also known as soft scalability). This method was developed by Sandia National Laboratories, Albuquerque, New Mexico, and has been used on many projects. However, there are some serious problems with this approach. The first is, as noted in the Section 5 of this appendix, that the approach makes assumptions that do not apply to this code; therefore, it is not a particularly appropriate metric to use in this case. The second problem is that the approach assumes that the current problem size is running efficiently and fast enough when N processors are used. If this is the case, then when a problem twice the size is confronted and 2N processors are used, one would still be happy with the performance. However, the approach fails to address the situation when the speed or efficiency for the current problem size is considered to be unacceptable. The third problem is that the approach assumes that one really wants to be running a bigger problem. If that is not the case, then scaled speedup is of little or no value even in cases when the other two objections do not apply.

The fourth and final possibility that we will consider is the case of fixed size speedup. The proponents of scaled acceleration will point out that for large enough numbers of processors this will always result in problems with both Amdahl's Law and with a poor ratio between communication and computation. They are absolutely right. However, if one has a problem with a limited level of parallelism, these objections may or may not be important. Continuing, it is natural to want the performance to double when the number of processors being used is doubled. In fact, when confronting problems involving large amounts of parallelism, this is the ideal behavior. It is also the behavior that will help one achieve the best use of the hardware (all things being equal). However, in the case of codes with a high level of algorithmic efficiency and serial efficiency, it may be reasonable to relax this requirement slightly. In particular, in Section 5 of this appendix, it was shown that the ideal behavior for codes with limited levels of parallelism is a stair-stepping behavior. Therefore, we suggest that if the stair-stepping behavior can be adequately predicted, that behavior should be used as the basis for this metric. In cases when this cannot be done, one might want to slightly adjust the acceptable level of parallel efficiency (e.g., decreasing it by 5% to 10% for 30 to 50 processors, and 10% to 20% for 50 to 100 processors). This still leaves the project responsible for the way the code interacts with the hardware, the effects of Amdahl's Law, and several other effects that will frequently result in less than ideal behavior. The one remaining sticking point is to determine things such as what problem sizes will be run and how many processors need to be used for each of the problems or problem sizes.

INTENTIONALLY LEFT BLANK

APPENDIX C

ZNSFLOW CHSSI PROJECT MILESTONES

INTENTIONALLY LEFT BLANK

ZNSFLOW CHSSI Project Milestones

The ZNSFLOW CHSSI project took place over several years. It is difficult to plan software development over a span of years. Some of the particular difficulties of software development are keeping pace with developing hardware and upgraded compilers, operating systems, and scalable computer models. These and other factors forced many modifications of the original ZNSFLOW development plan. Milestone charts provide documentation to measure the development of the ZNSFLOW software and progress of the overall CHSSI project and provide some insight for visualizing the development path of the final product. Following are milestone charts that provide a time table for the development of the ZNSFLOW CHSSI project.

	Milestone Item / Description	Software Definition		Scalable Implementation		Test & Calibrate		Beta Release		Final Release	
		Plan	Actual	Plan	Actual	Plan	Actual	Plan	Actual	Plan	Actual
1	Software Development:										
1.1	ZNS: Implicit Option	Apr-96	Apr-96	Feb-97	Aug-97	Aug-98	Oct-98	Jun-97	Oct-98	Mar-99	Jul-99
1.2	ZNS: DICE	Mar-97	Jun-97	Mar-98	Oct-97	May-98	May-98	Jun-98	Oct-98	Mar-99	Jul-99
2	Suite of Test Problems									Dec-97	Dec-97
3	Demostration Problems										
3.1	Guided MLRS Missile									Mar-99	Mar-99
3.2	BAT Separation from ATACM Missile									Nov-98	Nov-98
4	User Group Meetings									Mar-99	Aug-99
5	Documentation									Mar-99	Sep-99
6	Reports									Mar-99	Sep-99

Figure C-1. ZNSFLOW Milestone Schedule.

	Milestone Item / Description	Start		Complete	
		Plan	Actual	Plan	Actual
1	**Software Development:**				
1.1	Implicit ZNS:	Apr-96	Apr-96	Mar-99	Jul-99
1.1.1	Scalable Algorithm (SA)	Apr-96	Apr-96	Dec-97	Dec-97
1.1.1.1	SA:PCA-single node	Apr-96	Apr-96	Feb-97	Feb-97
1.1.1.2	SA:Origin2000 single node	Apr-97	Apr-97	Dec-97	Dec-97
1.1.1.3	SA:Origin2000 multiple node	Sep-97	Jan-98	Dec-97	Dec-97
1.1.1.4	SA:T3D	Apr-96	Apr-96	Jun-97	Jun-97
1.1.1.5	SA:T3E	Apr-97	Apr-97	Dec-98	Dec-98
1.1.2	Restructure Code (RC)	Aug-96	Aug-96	Jun-98	Jun-98
1.1.3	Pointwise Turbulence Model	Jun-97	Apr-98	Oct-98	Mar-99
1.1.4	Merge with DICE	Jan-97	Jan-97	May-98	May-98
1.1.5	Chimera (C)	Sep-96	Sep-96	Dec-97	Dec-97
1.1.5.1	C: Origin 2000	Sep-96	Sep-96	Sep-97	Sep-97
1.1.5.2	C:T3E	Nov-97			
1.1.9	Alpha Release	Jun-97	Jun-97	Sep-98	Sep-98
1.1.10	Beta Release	Jun-98	Jun-98	Sep-99	Sep-99

Figure C-2. ZNSFLOW Milestone Schedule Continued.

	Milestone / Item Description	Start		Complete	
		Plan	Actual	Plan	Actual
1.2	DICE Environment	Jan-97	Jan-97	May-98	Jul-98
1.2.1	common grid file structure	Mar-97	Mar-97	Sep-97	Sep-97
1.2.2	common q file structure	Jun-97	Jun-97	Dec-97	Dec-97
1.2.3	GUI for Implicit ZNS input	Jun-97	Jun-97	May-98	Jul-98
1.2.4	Visualization	Sep-96	Sep-96	May-98	Jun-98
1.2.4.1	Planes	Jan-97	Jan-97	Apr-97	May-97
1.2.4.2	Isosurfaces	Jan-97	Jan-97	Jun-97	Jun-97
1.2.4.3	Ensight Interface	Jan-97	Jan-97	May-98	Dec-97
1.2.6	Grid Generators	Sep-97	Sep-97	Oct-98	Dec-98
1.2.6.1	Genie	May-98	May-98	Oct-98	Dec-98

Figure C-3. ZNSFLOW Milestone Schedule Continued.

	Milestone / Item Description	Start		Complete	
		Plan	Actual	Plan	Actual
2	Test and Evaluation				
2.1	Suite of graded test problems	Apr-96	Apr-96	Dec-97	Dec-97
2.1.1	Uniform flow	Jan-97	Feb-97	Mar-97	Mar-97
2.1.2	Flat plate boundary layer	Feb-97	Feb-97	Apr-97	Mar-98
2.1.3	KTA missile	Apr-96	Apr-96	Sep-97	Sep-97
2.1.4	Spherical Nose Cap	Jun-96	Jun-97	Sep-97	Sep-97
2.1.5	SOCBT	Feb-97	Apr-97	Jul-97	Jul-97
2.2	Validation plan	Apr-96	Apr-96	Sep-97	Apr-98

Figure C-4. ZNSFLOW Milestone Schedule Continued.

	Milestone / Item Description	Start		Complete	
		Plan	Actual	Plan	Actual
3	Demonstration Problems				
3.1	Guided MLRS Missile	Apr-98	Sep-98	Mar-99	Mar-99
3.1.1	Problem definition	Apr-98	Sep-98	Sep-98	Sep-98
3.1.2	Grid definition	Apr-98	Sep-98	Sep-98	Sep-98
3.1.3	Calculation on Origin 2000	Dec-98	Dec-98	Jan-99	Mar-99
3.2	BAT projectile	Apr-96	Apr-96	Nov-98	Sep-98
3.2.1	Problem definition	Apr-96	Apr-96	May-96	May-96
3.2.2	Grid definition	Dec-96	Dec-96	Jan-97	Jan-98
3.2.3	Calculation on Origin 2000	Jun-97	Jan-98	Apr-98	Sep-98

Figure C-5. ZNSFLOW Milestone Schedule Continued.

	Milestone / Item Description	Start		Complete	
		Plan	Actual	Plan	Actual
4	4. User group meetings:				
4.1	4QFY96			Aug-96	Aug-97
4.2	3QFY97			Aug-97	Oct-97
4.3	3QFY98			Aug-98	Oct-98
4.4	3QFY99			Mar-99	Aug-99
5	Documentation:				
5.1	ARL CHSSI home page	Sep-97	Jan-98	Mar-99	Mar-99
5.2	ZNS user/technical manuals	Jun-96	Jun-97	Mar-99	Mar-99
5.3	Establish training classes	Aug-98	Oct-98	Mar-99	Aug-99
6	Reports:				
6.1	FY96: end year			Nov-96	Nov-96
6.2	FY97: mid year			Jun-97	Jul-97
6.3	FY97: end year			Nov-97	Dec-97
6.4	FY98: mid year			Jun-98	Jun-98
6.5	FY98: end year			Nov-98	Jan-99
6.7	FY99: Final Report			Mar-99	Sep-99

Figure C-6. ZNSFLOW Milestone Schedule Continued.

INTENTIONALLY LEFT BLANK

NO. OF COPIES	ORGANIZATION
2	ADMINISTRATOR DEFENSE TECHNICAL INFO CENTER ATTN DTIC OCP 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218
1	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL CS AL TA REC MGMT 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL CI LL TECH LIB 2800 POWDER MILL RD ADELPHI MD 207830-1197
1	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL DD 2800 POWDER MILL RD ADELPHI MD 20783-1197
7	CDR US ARMY ARDEC ATTN AMSTE AET A R DEKLEINE C NG R BOTTICELLI H HUDGINS J GRAU S KAHN W KOENIG PICATINNY ARSENAL NJ 07806-5001
1	CDR US ARMY ARDEC ATTN AMSTE CCH V PAUL VALENTI PICATINNY ARSENAL NJ 07806-5001
1	CDR US ARMY ARDEC ATTN SFAE FAS SD MIKE DEVINE PICATINNY ARSENAL NJ 07806-5001
2	USAF WRIGHT AERONAUTICAL LABS ATTN AFWAL FIMG DR J SHANG MR N E SCAGGS WPAFB OH 45433-6553
3	AIR FORCE ARMAMENT LAB ATTN AFATL/FXA STEPHEN C KORN BRUCE SIMPSON DAVE BELK EGLIN AIR FORCE BASE FL 32542-5434

NO. OF COPIES	ORGANIZATION
1	CDR NSWC CODE B40 DR W YANTA DAHLGREN VA 22448-5100
1	CDR NSWC CODE 420 DR A WARDLAW INDIAN HEAD MD 20640-5035
1	CDR NSWC ATTN DR F MOORE DAHLGREN VA 22448
1	NAVAL AIR WARFARE CENTER ATTN DAVID FINDLAY MS 3 BLDG 2187 PATUXENT RIVER MD 20670
4	DIR NASA LANGLEY RESEARCH CENTER ATTN TECH LIBRARY MR D M BUSHNELL DR M J HEMSCH DR J SOUTH LANGLEY STATION HAMPTON VA 23665
2	ARPA ATTN DR P KEMMEY DR JAMES RICHARDSON 3701 NORTH FAIRFAX DR ARLINGTON VA 22203-1714
6	DIR NASA AMES RESEARCH CENTER T 27B-1 L SCHIFF T 27B-1 T HOLST MS 237-2 D CHAUSSEE MS 269-1 M RAI MS 200-6 P KUTLER MS 258 1 B MEAKIN MOFFETT FIELD CA 94035
2	USMA DEPT OF MECHANICS ATTN LTC ANDREW L DULL M COSTELLO WEST POINT NY 10996

NO. OF COPIES	ORGANIZATION
2	UNIV OF CALIFORNIA DAVIS DEPT OF MECHANICAL ENGRG ATTN PROF H A DWYER PROF M HAFEZ DAVIS CA 95616
1	AEROJET ELECTRONICS PLANT ATTN DANIEL W PILLASCH B170 DEPT 5311 PO BOX 296 1100 WEST HOLLYVALE STREET AZUSA CA 91702
1	MIT TECH LIBRARY 77 MASSACHUSETTS AVE CAMBRIDGE MA 02139
1	GRUMANN AEROSPACE CORP AEROPHYSICS RESEARCH DEPT ATTN DR R E MELNIK BETHPAGE NY 11714
2	MICRO CRAFT INC ATTN DR JOHN BENEK NORMAN SUHS 207 BIG SPRINGS AVE TULLAHOMA TN 37388-0370
1	LANL ATTN MR BILL HOGAN MS G770 LOS ALAMOS NM 87545
1	METACOMP TECHNOLOGIES INC ATTN S R CHAKRAVARTHY 650 HAMPSHIRE ROAD SUITE 200 WESTLAKE VILLAGE CA 91361-2510
2	ROCKWELL SCIENCE CENTER ATTN S V RAMAKRISHNAN V V SHANKAR 1049 CAMINO DOS RIOS THOUSAND OAKS CA 91360
1	ADVANCED TECHNOLOGY CTR ARVIN/CALSPAN AERODYNAMICS RESEARCH DEPT ATTN DR M S HOLDEN PO BOX 400 BUFFALO NY 14225

NO. OF COPIES	ORGANIZATION
1	UNIV OF ILLINOIS AT URBANA CHAMPAIGN DEPT OF MECH & IND ENGINEERING ATTN DR J C DUTTON URBANA IL 61801
1	UNIVERSITY OF MARYLAND DEPT OF AEROSPACE ENGRG ATTN DR J D ANDERSON JR COLLEGE PARK MD 20742
1	UNIVERSITY OF NOTRE DAME DEPT OF AERONAUTICAL & MECH ENGRG ATTN PROF T J MUELLER NOTRE DAME IN 46556
1	UNIVERSITY OF TEXAS DEPT OF AEROSPACE ENGRG MECH ATTN DR D S DOLLING AUSTIN TX 78712-1055
1	UNIVERSITY OF DELAWARE DEPT OF MECHANICAL ENGRG ATTN DR JOHN MEAKIN NEWARK DE 19716
4	COMMANDER USAAMCOM ATTN AMSAM RD SS AT ERIC KREEGER GEORGE LANDINGHAM CLARK D MIKKELSON ED VAUGHN REDSTONE ARSENAL AL 35898-5252
4	LOCKHEED MARTIN VOUGHT SYS PO BOX 65003 M/S EM 55 ATTN PERRY WOODEN W B BROOKS JENNIE FOX ED MCQUILLEN DALLAS TX 75265-0003
1	COMMANDER US ARMY TACOM-ARDEC BLDG 162S ATTN AMCPM DS MO PETER J BURKE PICATINNY ARSENAL NJ 07806-5000
1	DIR NASA LANGLEY RESEARCH CENTER MS 499 P BUNING HAMPTON VA 23681

NO. OF COPIES	ORGANIZATION
	ABERDEEN PROVING GROUND
2	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL CI LP (TECH LIB) BLDG 305 APG AA
3	CDR US ARMY ARDEC FIRING TABLES BRANCH ATTN R LIESKE R EITMILLER F MIRABELLE BLDG 120
1	DIR USARL ATTN AMSRL CI N RADHAKRISHNAN BLDG 394
4	DIR USARL ATTN AMSRL CI H D HISLEY D PRESSEL C ZOLTANI C NIETUBICZ BLDG 394
1	DIR USARL ATTN AMSRL CI H W STUREK BLDG 328
2	DIR USARL ATTN AMSRL WM I MAY L JOHNSON BLDG 4600
2	DIR USARL ATTN AMSRL WM B A W HORST JR W CIEPIELLA BLDG 4600
1	DIR USARL ATTN AMSRL WM B E M SCHMIDT BLDG 390A
7	DIR ARL ATTN AMSRL WM BA W D'AMICO F BRANDON T BROWN L BURKE J CONDON B DAVIS M HOLLIS BLDG 4600

NO. OF COPIES	ORGANIZATION
20	DIR USARL ATTN AMSRL WM BC P PLOSTINS M BUNDY G COOPER H EDGE (5 CYS) J GARNER B GUIDOS K HEAVEY D LYON A MIKHAIL V OSKAY J SAHU K SOENCKSEN D WEBB P WEINACHT S WILKERSON A ZIELINSKI BLDG 390
1	DIR USARL ATTN AMSRL WM BD B FORCH BLDG 4600
2	DIR USARL AMSRL WM BE M NUSCA J DESPIRITO BLDG 390
1	DIR USARL ATTN AMSRL WM BF J LACETERA BLDG 120
1	DIR USARL ATTN AMSRL WM TB R LOTTERO BLDG 309
	ABSTRACT ONLY
1	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL CS AL TP TECH PUB BR 2800 POWDER MILL RD ADELPHI MD 20783-1197

INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 2000	Final

4. TITLE AND SUBTITLE
Common High Performance Computing Software Support Initiative (CHSSI) Computational Fluid Dynamics (CFD)-6 Project Final Report: ARL Block-Structured Gridding Zonal Navier-Stokes Flow (ZNSFLOW) Solver Software

5. FUNDING NUMBERS
PR: 1L162628AH80

6. AUTHOR(S)
Edge, H.L.; Sahu, J.; Heavey, K.R.; Weinacht, P.; Sturek, W.B.; Pressel, D.M.; Zoltani, C.K.; Nietubicz, C.J. (all of ARL); Clarke, J. (Raytheon Systems); Behr, M. (Rice University); Collins, P. (Department of the Treasury)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
Weapons & Materials Research Directorate
Aberdeen Proving Ground, MD 21010-5066

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
Weapons & Materials Research Directorate
Aberdeen Proving Ground, MD 21010-5066

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
ARL-TR-2084

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report presents an overview of the software developed under the common high performance computing software support initiative (CHSSI), computational fluid dynamics (CFD)-6 project. Under the project, a zonal Navier-Stokes flow solver tested and validated via years of productive research at the U.S. Army Research Laboratory was rewritten for scalable parallel performance on both shared memory and distributed memory high performance computers. At the same time, a graphical user interface (GUI) was developed to help the user set up the problem, provide real-time visualization, and execute the solver. The GUI is not just an input interface but provides an environment for the systematic, coherent execution of the solver, thus making it a more useful, quicker and easier application tool for engineers. Also part of the CHSSI project is a demonstration of the developed software on complex applications of interest to the Department of Defense (DoD). Results from computations of 10 brilliant antitank (BAT) submunitions simultaneously ejecting from a single Army tactical missile and a guided multiple launch rocket system missile are discussed. Experimental data were available for comparison with the BAT computations. The CFD computations and the experimental data show good agreement and serve as validation for the accuracy of the solver. The software has been written with large memory requirements and scalability in mind. For a grid size of 59 million points, the performance achieved on an Silicon Graphics, Incorporated, Origin 2000 with 96 processors is 18 times the performance that could be achieved via a computer with the processing speed of a single Cray C-90 processor.

14. SUBJECT TERMS
CHSSI; high performance computing; Navier-Stokes; computational fluid dynamics; missiles

15. NUMBER OF PAGES
80

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	